Orthogonal Recursive Bisection on the GPU for Accelerated Load Balancing in Large N-Body Simulations

-

Bachelor Thesis

Andrin Rehmann

July 31, 2022

Contents

1 Introduction

The N-Body technique has been used for decades to simulate the Universe to compare theory with observations. It uses ”particles” to represent objects of a certain mass and computes the forces by applying the gravity equation. As the forces operate over infinite distance it is necessary to consider all pair-wise interactions, making a naive implementation $\mathcal{O}(n^2)$. Clearly this does not scale with large particle counts.

A common solution is to partition the space, in which the particles are contained, into a set of subspaces, also called cells. These cells are then stored in a space partitioning tree data structure (SPTDS) in order to speed up the simulation with the Fast Multipole Method (FMM) to $O(n)$. Building the data structure uses a significant percentage of the overall simulation time. As of now this was usually done on the CPU, not leveraging GPU acceleration. Tree walking and building algorithms heavily rely on control statements such as if / else. GPU programming in contrast, benefits from large packable data arrays without any branch divergence and thus special attention has to be paid to the exact implementation details. In this thesis we establish upper limits for the speedup of GPU accelerated SPTDS building algorithms over its CPU only counterpart. We then propose, implement and analyze an efficient way of building the SPTDS on the CPU and on the GPU.

TODO: Use past tense here?

TODO: Describe measurement results here

2 Background

The Orthogonal Recursive Bisection method constructs a SPTDS where all resulting cells encompass subspaces similar to cubes. K-d construction methods generate SPTDS as well, but shapes of the resulting subspaces can be strongly skewed and are ill suited for FMM as explained in more detail in section 2.1. Both algorithms start with a single cell and iteratively construct the SPTDS by searching for axis aligned planes cutting the volume of each leaf cell into two. From the resulting two volumes, two new cells are constructed and appended to the SPTDS.

In k-d tree construction, the axis lying orthogonal to the cut plane is chosen periodically, meaning all cells to be found on the same level of the SPTDS are cut with a plane orthogonal to the same axis. In the case of OBR, the axis where a cells volume is largest is picked to compute an optimal plane. Meaning cells on the same level of a tree can have cut planes of variable orientations.

The resulting tree, a SPTDS, is used to perform a multipole expansions of each cell or subspace to approximate the forces acting between the particles. A mutlipole expansion is essentially a mathematical representation of a group of objects as a single function and is explained in more detail in section 2.1. Multipole expansions reduces the complexity of gravity calculations from $O(n^2)$ to $\mathcal{O}(n \log n)$. More recently the Fast Multipole Method (FMM), also relying on the SPTDS, gained wider adoption. This technique further reduces the complexity to $\mathcal{O}(n)$.

The computational effort required in modern astrophysical N-Body simulations can be divided into three categories:

1. **Global Tree Building / Load Balancing:** In supercomputers, particles are distributed equally among computing nodes to leverage node level parallelization. ORB is used to generate a well suited SPTDS for FMM on a node level.

2. **Local Tree Building:** The same approach is repeated on a local level, where instead of distributing the workload among nodes, its distributed among threads. The tree from step 1. is expanded by the locally computed tree.

3. **Force Calculation and Integration:** The STPDS constructed in step 1. and 2. combined with FMM is used to compute force integration.

Before the implementation of FMM into codes, and particularly before the era of modern accelerated computing (e.g., SIMD vectorization and GPU computing) the forces calculations (3.) dominated the computational cost. In more recent simulations, each category is about one third of the total calculation time[1]. Making Global Tree Building (2.) and Local Tree Building (1.) subjects to possible performance improvements, as GPU acceleration is not exploited.

We propose to implement Global Tree Building using CUDA to accelerate load balancing. As the same problem is very closely related to local tree building, the results may also be used to accelerate part (3.) of the simulation.

Our main objective is to improve the runtime of astrophysical simulations, or more specifically PKDGrav. Simultaneously penalties with regards to N, the total number of particles, should be kept as low as possible. Simulation accuracy benefits from large N and from reduced runtime as a reduced runtime can result in an increased number of time steps on the same computing resources. As GPU acceleration are low level and hardware specific details is relevant, we want to adapt and optimize the algorithms with regards to the target computer Piz Daint.

2.1 Fast Multipole Expansion

In mathematics, a multipole is a concept used to approximate the shape of a physical body. In our case the physical body are the particles contained within the volume of a cell. The multipole expansion is the mathematical series that results from this approximation. The resulting series can simplify the original body, by choosing a suitable precision.

To keep force integration within a reasonable error margin, in most cases it is not necessary to compute all n to n interactions. Particles far away are instead grouped and summarized with a multipole expansion. A group in this context corresponds to a cell in a SPTDS.

On a more intuitive level, one can imagine a large cluster of objects far away from planet Earth. If we wish to compute the interactions between every single object of the cluster and Earth, we will need to perform $O(n^2)$ computations. However, since the object is reasonably far away, it does not make sense to compute every interaction. Instead, we summarize the cluster into a single object. We can then simply compute the gravity acting between planet Earth and the group. Consequently, objects within our own solar system would be too close to planet Earth to approximate its effects using a multipole. In such cases, it is necessary to perform all $O(n^2)$ computations. Note that this example does not directly translate to the simulations, as a particle is not necessarily equivalent to a planet or a star. More on that in section 2.2.

2.1.1 Advantage of Cubic Bounding Volumes

As mentioned, when building the SPTDS with ORB, we search for a median in the axis where the cell volume is largest. It is assumed that this is the best method to build cubic shapes on all levels of the tree, as also intermediate cell volumes are leveraged by FMM.

From [1]: "the opening radius of a cell is given by some constant factor times r_{max}". r_{max} is the distance between the center of mass of a cell and its most distant corner. The opening radius correlates with the error ratio, which in turn influences how many timesteps have to be performed. Of the family of rectangular cuboids, the cube is the shape where the average distance between any point and its most distant corner point is the smallest.

2.2 Target Data and Notation

A modified random Gaussian distributed is used to generate the target dataset, where its set to fit measured constraints of the actual universe. Thus the number of particles can set dynamically. In the simulation, a single particle has a mass of 10^8 solar masses.

- We define a particle as par_i where $i \in \{0, ..., N\}$.
- We define the space of binary numbers with a precision of p as $\mathbb{B}_p$ where we have $a \in \mathbb{B}_p \Leftrightarrow a \in \{0, 1\}^p$
- We define the corner coordinates of root domain with $\vec{lower}, \vec{upper}$ where we have $\vec{lower}, \vec{upper} \in \mathbb{B}^3_p$.
- We define the coordinates of a particle par_i as $\vec{x_i}$ for which holds $\{\vec{x} | \vec{lower} \leq \vec{x} \leq \vec{upper}, \vec{x} \in \mathbb{B}^3_p\}$. When refering to a single coordinate of a particle object we do so by writing $\vec{x}_{i,j}$. Finally the array of all particles positions in a single axis is denoted as $\vec{x}_{:,j}$.

It lies in the nature of the universe, that large clusters of particles are found in some places, whereas other areas may be vastly empty of any objects. This is a very crucial characterization and differentiates this dataset from many other application of FMM, where the distribution are more uniform.

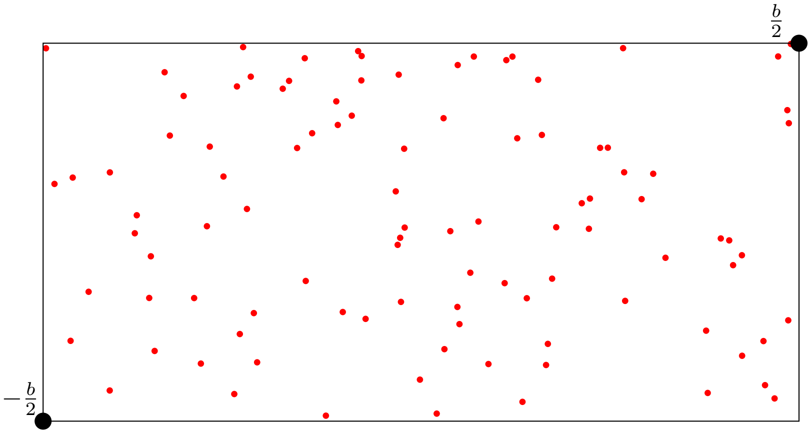


Figure 1: Uniform random distribution of 3D coordinates in cube domain projected onto a 2d plane

2.3 Procedure Overview

When a simulation is initialized, all data is distributed randomly among the nodes. With regards to particle positions, the target data is initially unstructured, therefore we have no way distribute the particles in a better way. As we want to leverage FMM we need to compute a SPTDS across all nodes.

The SPTDS has to be recomputed each time after force integration is computed. Some particles may have crossed their cell boundaries during the process, violating the very constraint that all particles "owned" by a cell are contained within its volume. There are smarter ways to update the tree, which include more relaxed constraints, we will not focus on updating strategies and simply recompute the entire tree after each time step.

Depicted below is an illustration giving a high level overview over the necessary steps required to perform the load balancing.

1. The simulation data is generated and located on a central storage device.

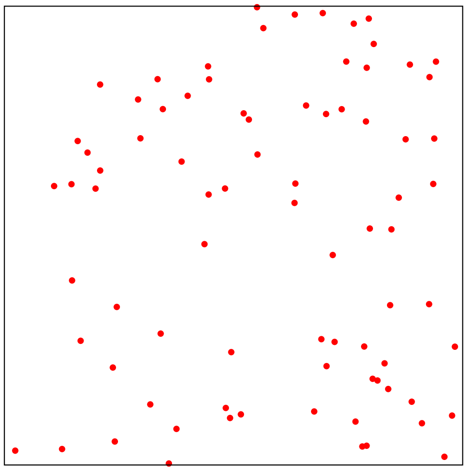


Figure 2: Initial data with particles

2. Since the data is initially unstructured, is is distributed randomly among the nodes in the super computer.

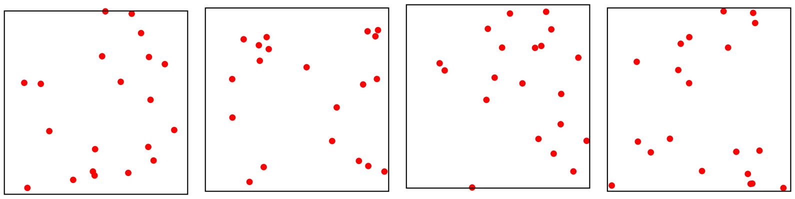


Figure 3: Particles are randomly distributed among 4 nodes

3. The SPTDS is constructed with ORB. The tree is computed globally

across all nodes, meaning the volumes of the individual cell are identical for all nodes.

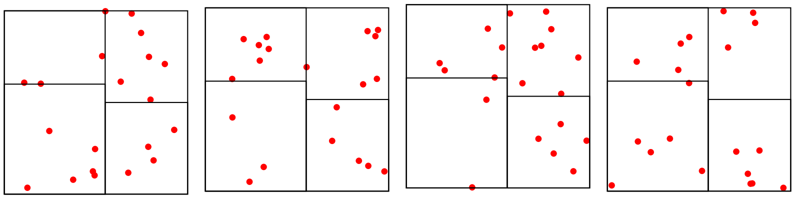


Figure 4: Space partitioning tree data structure is computed

4. In a final step the particles are redistributed among the nodes, such that each node stores all particles contained in the volume of a unique cell.

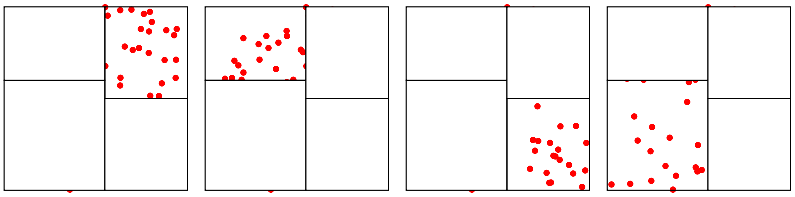


Figure 5: Leaf cells of data structure are distributed among nodes

2.4 PKDGrav

3 Related Work

A number of GPU accelerated k-d tree construction algorithms exists, in most cases the methods cannot be translated due to different axis choosing method. Fore example in the master thesis of Voglsam [2] a k-d tree is computed on the GPU and used to improve ray tracing for 3D graphics. However it makes use of a binning algorithm to

ORB can be subdivided into the following steps: **TODO:** LOD??

1. Cut cells on last level in tree
   - (a) Make cut plane position guess
   - (b) Count number of particles left to the cut plane
   - (c) Repeat 1. till correct plane was found
2. Partition particles
3. Repeat till desired depth was reached

Thrust by NVIDIA [3] has an implementation of a reduction, which can be used to perform step 1.b), a binary search is equivalent to the entire step 1. and finally it exposes a partition interface. The library is very high level and does not allow us to control memory operations, which are very crucial to a performant implementation of ORB. Furthermore CUB [4], also developed by NVIDIA, exposes a more low level API of such elements but the implementation is not general enough to deal with some performance issues. As the SPTDS grows, so do the number of leaf cells and the number of cells for which a cut plane has to be found. If we initialize a kernel for each cell, the number of kernels initialized grow exponentially, resulting in kernel invocation overhead, which dominates the actual computations.

So far there exists no...

Not sure how this works exactly: $https://github.com/johnarobinson77/KdTreeGPU$

Min Max binning. Probably does not work when axes are not switched periodically. $https://www.cg.tuwien.ac.at/research/publications/2013/Voglsam_2013_RRT/Voglsam_2013_RRT-Thesis.pdf$ and the paper: $https://jcgt.org/published/0004/01/03/paper.pdf$ Lots of GPU memory overhead.

mass assignment

TODO: Write how ORB is similar to quicksort, how individual parts could be done using thrust library.

4 Orthogonal Recursive Bisection (ORB)

This section explains ORB, which is used to build a SPTDS with non cycling axis choices. Each cell in the tree is enriched with the following properties:

- V_{cell} 3D volume where all corresponding particles are contained within.
- d_{cell} Number of leaf cells to be found among all subsequent cells.
- N_{cell} Number of particles encompasses in the volume of the cell.

Naturally each cell has two pointers pointing to its two children, which are used to traverse the tree, parent cell pointers are not of importance but could be added as well.

4.1 Memory and Workload Balancing

As mentioned we want to (1) improve the runtime of ORB whilst (2) at the same time keeping the number of particles N as large as possible. There exists some averse effects when disregarding interactions between minimizing runtime and maximizing N.

To reduce the force integration error across all particles, some particles require more timesteps than others. This is caused by the vastly variant forces which are exerted on the particles. Whilst some follow a straight path with an almost constant velocity requiring little computational effort for great precision, others are influenced by strong gravitational poles resulting in highly curved movement paths and correlating with higher numbers of timesteps required.

We denote the workload for a particle p_i as the weighting function $w(p_i)$. The workload correlates with the number of simulations steps, that need to be computed for a single particle. In an optimally parallelized system balanced in terms of computational effort, the workload should be very similar among computing units. Where ideally the sum over all the particles considered by a thread is equal to the sum of all others. With strongly varying weights among particles, perfectly balancing the computational effort, results in a unbalanced distribution in terms of memory.

Let us consider a simple example to illustrate the point. Given the set of particles $A = \{p_1, p_2, .., p_{2m/3}\}$ and respectively $B = \{p_{2m/3+1}, p_2, .., p_m\}$. This yields a total number of particles equivalent to $N = m$. We assuming that $\forall p \in A : w(p) = 1$ and $\forall p \in B : w(p) = 2$. We assign all particles from set A to process with rank 0 and the particles from B to rank 1. It follows $\sum_{p \in A} w(p) = \sum_{p \in B} w(p)$. Thus the two processors are balanced in terms of computing costs, but not in terms of memory size. In fact, process with rank 0 has $2m/3$ elements and rank 1 has $m/3$ elements. Assuming each process has a memory size of $2m/3$, then clearly this configuration is not optimal. If we were to favor memory balancing, we could assign $2m/3$ to each processor and we would be able process $(4/3) \times m$ particles in total, which is larger than the original N and therefore favorable since a dataset with a larger N could be used.

To which degree its ideal favor memory over workload balancing is difficult to answer. For now parametrize the workload using the weighting function. If we decide to completely ignore the workload balancing and solely focus on memory balancing, we can simply set $w(p_i) = 1$ for all particles.

4.2 ORB

To introduce the ORB algorithm, we consider a recursive implementation of ORB as it is easier to understand than the iterative version. We define all cell properties recursively, allowing us to describe the algorithm in detail and derive the necessary constraints.

As we initialize our recursion with the root cell alone, it follows that V_{root} is equivalent to entire volume of the input data. $d_{cell} = d$ and $N_{root} = N$ are true per definition of N and d. When a cell is cut into two child cells, we refer to the children as $leftCell$ and $rightCell$. For any cell the following equations hold:

$$V_{cell} = V_{leftCell} \cup V_{rightChild} \tag{1}$$

$$V_{leftCell} \cap V_{rightChild} = \emptyset \tag{2}$$

Defining d_{cell} for each cell is especially important, since in some cases d is a non power of two number, meaning some recursion paths terminate earlier. The larger portion of leaf cells are always allocated to the left cell, as this ensures that the final constructed tree is a nearly complete binary tree.

$$d_{leftCell} = \left\lceil \frac{d_{cell}}{2} \right\rceil \tag{3}$$

$$d_{rightCell} = d_{cell} - d_{leftCell} \tag{4}$$

Trivially $d_{leftCell} - d_{rightCell} \leq 1$ holds. **TODO:** Missing argumentation why its a nearly complete binary tree!!

Finally the number of particles encompassed in $V_{leftCell}$ and $V_{rigtCell}$ are defined as follows:

$$N_{leftCell} = min\left\{ x \in \{0, ..., N_{cell}\} : \sum_{i=0}^{x} w(p_i) \geq \frac{d_{leftCell}}{d_{cell}} \times \sum_{i=0}^{N_{cell}} w(p_i) \right\} - 1 \tag{5}$$

$$N_{rightCell} = N_{cell} - N_{leftCell} \tag{6}$$

To simplify the visual and numeric explanation of the ORB algorithm, we assume $\forall i \in \{0, .., N\} : w(p_i) = 1$.

As the cut plane, which divides $V_{leftCell}$ and $V_{rightCell}$ is axis aligned, searching for a single value c is sufficient to find an ideal construction. Its is considered ideal if the number of particles where $x_{i,a} \leq c$ are equivalent to $N_{leftCell}$.

After a plane is found, we can divide or cut the V_{cell} in two volumes, where $V_{leftCell}$ and $V_{leftCell}$ are constrained by the original V_{cell} and the cut plane. As defined in equation 2 the volumes do not intersect, thus $V_{leftCell}$ along axis a ends at c, where $V_{leftCell}$ start at c.

4.2.1 Algorithm

Algorithm 1 is the main routine of the orb algorithm. All the volumes are described using a lower and an upper boundary point, sufficiently describing a box, thus the only input parameters required are is x storing the particles positions, *lower*, *upper* and finally d. If d is equal to 1, this means the reduction must not be continued, as the target of a single subsequent leaf cell is already reached, since the cell itself is treated as a leaf cell. The stopping condition is formulated on lines 2-4. Line 6 describes a call to a method called $maxIndex()$, which essentially compares all provided values and returns the index of the maximum values. The result i provides us with the axis where the cell volume is largest. $d_{leftCell}$ is computed as described in equation 3 and its results is stored in a variable named d'. All prerequisites are met to compute the actual cut with the *cut* method which we will explain in more detail in section 4.4. The return value stored in *cut* is equivalent to the position of the cut plane along the cut axis. Using the *cut* value, the array of particles is partitioned as described in 4.5 returning the *mid* value which is the pivot index of the partitioning. Finally we can divide the original volume into two volumes as described on lines 10-13 and recursively call ORB method where we provide it with two slices of the x array.

Algorithm 1 The ORB main routine

```
procedure ORB($x$, $\vec{lower}, \vec{upper}, d$)
    if d = 1 then
        return                                                    ▷ Stopping condition
    end if
    $\vec{size} = \vec{upper} - \vec{lower}$
    $i = maxIndex(\vec{size}_0, \vec{size}_1, \vec{size}_2)$                ▷ Get index of max value

    $d' = \lceil \frac{d}{2} \rceil$
    $cut = cut(x, \vec{lower}_i, \vec{upper}_i, i, \frac{d'}{d})$                  ▷ Find cut plane
    $mid = partition(x, split, axis)$                             ▷ Partition particles

    $\vec{upper}' = \vec{upper}$
    $upperChild_i = cut$
    $\vec{lower}' = \vec{lower}$
    $lowerChild_i = cut$

    ORB($x_{0:mid,:}$,$\vec{lower}, \vec{upper}', d'$)
    ORB($x_{mid:len(x),:}\vec{lower}', \vec{upper}, d - d'$)
end procedure
```

4.3 By Example

A sample dataset is proposed to help with visual and numerical explanations of the algorithm. All particles in the sample dataset are assumed to have a weight of 1.

We explore the algorithm visually using the provided example dataset:

x	y	id
0.4	0.3	0
0.2	0.6	1
0.8	0.9	2
0.6	0.5	3
0.3	0.8	4
0.7	0.1	5
0.9	0.3	6

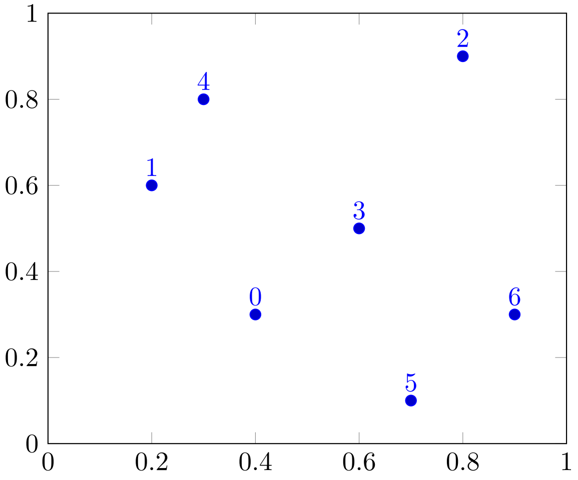


Figure 6: Example distribution with $N = 7$

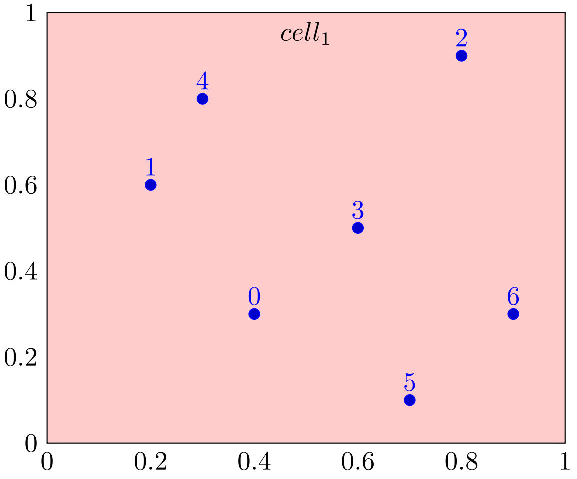


Figure 7: Example particles with ORB at recursion depth 0

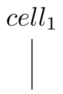


Figure 8: Tree with ORB at recursion depth 0

Depicted in figure 7 & 8 is the SPTDS after initialization: there is only one

$cell_1$ which is root and V_{cell_1} encompasses the entire domain which in this case is the rectangle ranging from 0 to 1 in both the x and y axis. d_{cell_1} is equivalent to 3 and N_{cell_1} is 7.

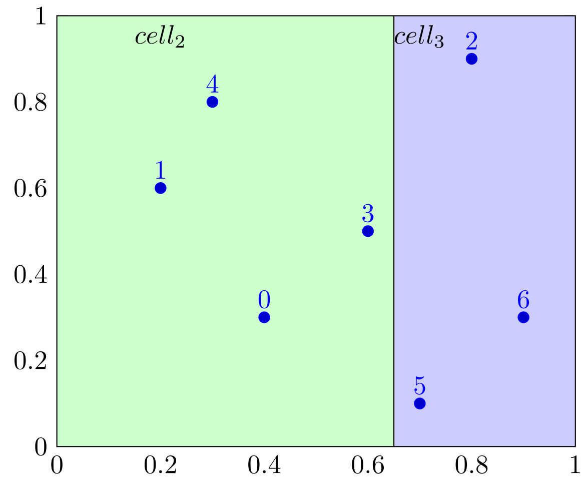


Figure 9: Example particles with ORB at recursion depth 1

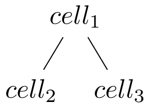


Figure 10: Tree with ORB at recursion depth 1

A cut plane, which translates to a cut line in simplified 2D example is constructed and $cell_2$ and $cell_3$ are generated accordingly.

9 & 10 Depict a SPTDS of depth 2, or two levels, where a cut plane was found dividing V_{cell_1} into V_{cell_2} and V_{cell_3}, $d_{cell_2} = 2$ and $d_{cell_3} = 1$ subsequently are calculated, meaning $cell_3$ is considered a leaf cell. Thus we can compute N_{cell_2} using a simplified version of equation 5 since we assume all weights are equal 1: $N_{cell_2} = \lfloor \frac{d_{cell_2}}{d_{cell_3}} * N_{cell_1} \rfloor = \lfloor \frac{2}{3} * 7 \rfloor = 4$ subsequently $N_{cell_3} = 3$.

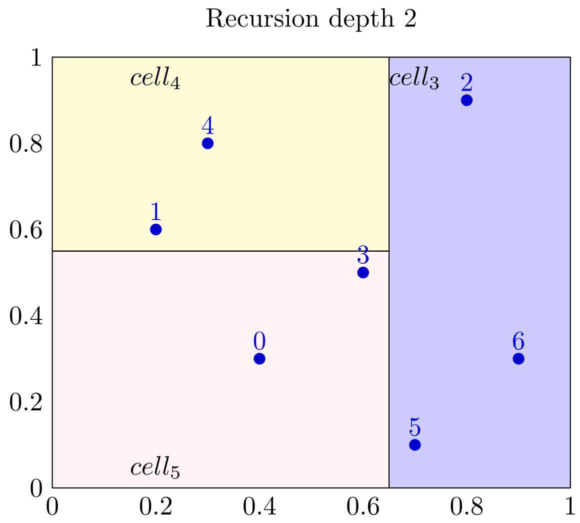


Figure 11: Example particles with ORB at recursion depth 2

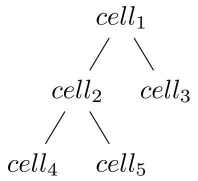


Figure 12: Tree with ORB at recursion depth 2

Again the procedure is repeated for $cell_2$ but not for $cell_3$ as there recursion is terminated by the stopping condition. Finally we end up with $cell_4$ and $cell_5$ where the stopping condition is met as well. The resulting SPTDS has 3 leaf cells and partitions the space into three subspaces.

4.4 Root finding

The *cut* algorithm takes an array of particle positions x, an $axis$, $left$ and $right$ boundaries and a $percentage$. Its goal is to return a position along the cut axis such that the particles less or equal to the a cut value are equivalent to the percentage multiplied by the length of the x array.

The problem is related to a selection algorithm and a quick select[5] could be used. However quick select has a worst case runtime of $O(n^2)$. Furthermore median algorithms could be explored as well, but they are mostly approximation algorithms with bad worst case runtimes. We can reformulate the problem as a root finding problem. To do so we define a function $f(c)$ which evaluates the number of elements smaller than the cut value minus half the number of total particles. When the function evaluates to zero, we have found an axis aligned cut plane, where exactly half the particles are located on the left side. There exist many different solvers for the root-finding problem, but the most stable and easiest to implement is the bisection method. Some solver combine approximate solver with the stable bisection method to generate fast but stable root finding methods. Such algorithms could be explored in later work.

Since we are operating on binary numbers with a limited precision, the bisection method is guaranteed to terminate after p steps.

4.4.1 Bisection Method

Initially an estimation for a cut is made, in this case the exact middle of the domain boundaries. Some median finding algorithms use improved guessing to speed up the process, again a method which could be leveraged in later work.

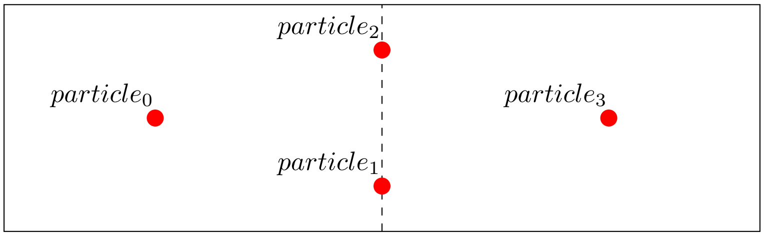
Because the maximal number of iterations is known, a loop can be used as seen in algorithm 2 on line 4. A cut is then computed (line 5), which in the first iteration is the center position between left and right. We then check weather the stopping condition has already been reached (line 7-9), meaning the cut lies within α points of the ideal result. If it was not reached the boundaries can be improved as follows: In case there are too many elements left of the cut, we know that the cut position was chosen too far to the right and we can be sure that the ideal cut must be left of the current guess. Therefore we adjust the boundaries, in this case we set *righ* equal to *cut* as seen on line 13. The analog concept can be applied in the other case (line 11).

Algorithm 2 Bisection Method

```
procedure CUT($x, left, right, axis, percentage$)
    $nLeft = 0$
    $cut = 0$

    for $k \in 0, .., p$ do
        $cut = (right + left)/2$
        $nLeft \leftarrow sum(x_{:,axis} < cut)$        ▷ Counting particles left of cut

        if $abs(nLeft - len(x)) * percentage < \alpha$ then
            Break                                        ▷ Stopping condition
        end if

        if $nLeft \leq len(x) * percentage$ then
            $left = cut$
        else
            $right = cut$
        end if

    end for
    return $cut$
end procedure
```

4.4.2 Edge Cases

Let us consider the following example particle distribution where $particle_1$ and $particle_2$ have identical x coordinates.



In this case there exists no ideal cut in the x (horizontal) axis. As either there is 1 particle to the left or 3, but no cut can result in 2 particles to the left. If we keep adding particles along the same line, the method performs even worse. However since the algorithm uses a deterministic for loop, the algorithm will terminate anyways.

QUESTION: Write this? seems too vague

4.4.3 Runtime Analysis

On line 5, the number of particles to the left of the cut plane are summed. As the array is unordered, all coordinates of one axis need to be read once, resulting in a runtime of $O(N)$. All other operations inside the for loop can be computed in a negligible constant time. The loop itself is repeated p number of times, where p corresponds to the precision of x coordinates.

To proof that the loop concludes after p iterations we assume integer numbers. With each iteration the range of possible solutions is divided by two. The same goes for integer numbers, each time a bit is removed, the size of the range of numbers which can be represented is reduced by two. Thus after p iterations the precision limit is reached and the cut cannot be improved.

4.5 Partition Algorithm

We want to continuously update the array storing the positions of the particles and in the words used before, have a direct correlation between x_i and i. This enables grouping particles within a cell in a fixed range of two indexes of the particles array. The advantages of this are two fold: Access all particles within a cell in constant time, manipulate particles within a cell using a slice of the array.

As seen in the pseudo code in algorithm 3, the algorithm looks for a pair of particles, where for both the coordinate along the relevant axis are on the wrong side of the cut plane. In this case the particles can be swapped (line 8) resulting in the correct position of both particles. We refer to [5] for a correctness proof and a more detailed explanation of the algorithm.

Algorithm 3 Partition Method

```
 1: procedure PARTITION($x, cut, axis$)
 2:     $i = 0$
 3:     for $k \in 0, ..N - 1$ do
 4:         if $\vec{x}_{k,axis} \leq cut$ then
 5:             while $\vec{x}_{i,axis} \leq cut$ and $i < N$ do
 6:                 $i = i + 1$
 7:             end while
 8:             $x_i, x_k = x_k, x_i$
 9:         end if
10:     end for
11:     $x_i, x_{N_j-1} = x_{N_j-1}, x_i$
12: end procedure
```

A runtime of $O(N)$ can be derived, as the algorithm iterates over all particles once. Since we need to touch each element at least once to partition the entire array there exists no better method.

Lets apply the algorithm to our running example. We start with our initial array of particles as follows:

x	**y**	**id**
0.4	0.3	0
0.2	0.6	1
0.8	0.9	2
0.6	0.5	3
0.3	0.8	4
0.7	0.1	5
0.9	0.3	6

We then partition the particles with a cut in the x axis set to 0.65 as seen in figure 9.

x	**y**	**id**
0.4	0.3	0
0.2	0.6	1
0.3	0.8	4
0.6	0.5	3
0.8	0.9	2
0.7	0.1	5
0.9	0.3	6

Finally we partition $cell_2$ into $cell_4$ and $cell_5$ with the cut position 0.55 along the y axis as seen in figure 9. We end up with the following array:

x	**y**	**id**
0.4	0.3	0
0.6	0.5	3
0.3	0.8	4
0.2	0.6	1
0.8	0.9	2
0.7	0.1	5
0.9	0.3	6

Note how all particles from $cell_4$ are contained in the range of 0-1. The particles of $cell_5$ in 2-3 and finally the particles contained in the volume of $cell_3$ can be found in 4-7.

5 Theoretical Analysis

The main goal of this thesis, is to improve the runtime of the ORB algorithm by leveraging the graphics processing unit over the central processing unit. Before implementing, it makes sense to explore if and how strong the performance benefits could in theory manifest itself. For this purpose we develop a simplified model, which we use to compare the CPU version against two different GPU variants. As GPU programming is very low level, the knowledge gained while developing the model also builds a solid theoretical foundation.

5.1 General Memory Model

For a consistent terminology of a computer, we briefly establish a general computing model, which can be applied to most modern high performance systems. We are looking at a single node and its hardware components, a supercomputer may have thousands of these computers linked together where different bandwidth are seen between individual nodes.

Both the CPU and the GPU have their own memory which are connected by a data link, where the the connections are bound by the memory bandwidths. We name the capacity of the CPU memory bandwidth B_{CPU} and the GPU memory bandwidth B_{GPU}. There is a separate data link between the CPU and the GPU memory, which is commonly referred to as PCI express or NVLink (for modern NVidia GPU‘s, in our model we use the term I_{GC}. In systems with multiple CPU there is a link between the individual CPU's which we denote as I_{CC}. Finally we call the link between GPUs as I_{GG}.

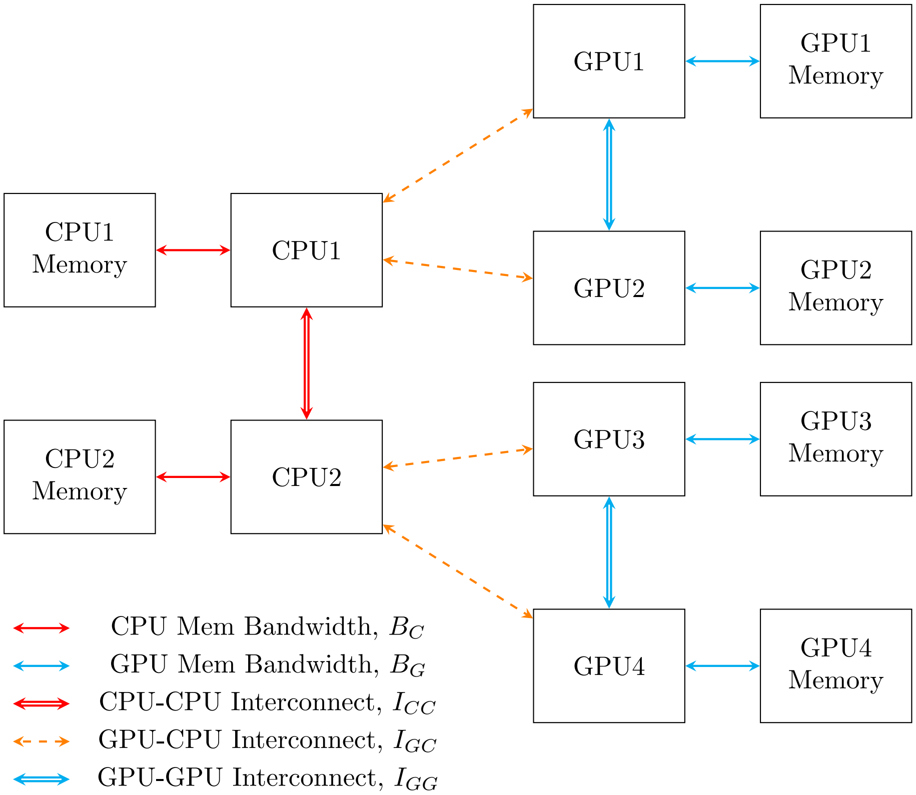


Figure 13: Illustration of the universal computing model

5.2 Supercomputers

For Piz Daint, Summit and Eiger we have collected all relevant hardware metrics and compiled it into the table seen in table 1. Piz Daint and Eiger were chosen, because we have the possibility to test the Code on its systems. Furthermore we include Summit as its is at the time of writing this thesis, one of the most capable supercomputers in the world.

Constant	Piz Daint [6]	Summit[7]	Alps (Eiger)
# Nodes	5704	4608	1024
# CPUs	1	2	2
CPU Model	Intel E5-2690 v3 [8]	IBM POWER9	AMD EPYC 7742[9]
CPU Mem.	64 GB	256 GB	??
B_C	68 GB/s	170 GB/s	204.8 GB/s x 2
I_{CC}	-	64 GB/s	??
Base GHZ_C	2.9 GHZ	4 GHZ	2.25 GHZ
Max GHZ_C	3.8 GHZ	4 GHZ	3.4 GHZ
# Cores	12	22	64
Architexture	Haswell	POWER9	AMD Infinity Architecture
SIMD	AVX2	?	AVX2
# GPUs	1	6	0
GPU Model	NVIDIA P100 [10]	NVIDIA V100s [11]	-
GPU Mem. Cap.	16 GB	16 GB × 6	-
B_G	732 GB/s	900 GB/s × 6	-
I_{GC}	32 GB/s	50 GB/s × 6	-
I_{GG}	-	50 GB/s	-
GPU Tflops	9,3	16.5	-
# CUDA Cores	3584	5120	-

Table 1: Datapoints of Supercomputers

5.3 Roofline Performance Model

In a first step we determine weather our computations are bound by memory bandwidth or the actual performance of the computing unit. The most costly computation is line 6 of the cut method (figure 2) with a runtime of $O(32 \times N)$. Oftentimes when an algorithm iterates over a large dataset performing only very little calculations on its individual elements, the limiting factor is the memory. To support this claim make roofline models for all three systems and check weather they are really memory bound. A roofline model compares arithmetic intensity in flops per byte against the actual performance of the computing chip in flops. Therefore we need to gather all the relevant data first.

5.3.1 Estimating Flops

For modern hardware, its fairly uncommon to release flops (floating point operations per second) values. Steadily evolving SIMD instruction sets result in varying performance for different implementation details, which in turn are compiled into different assembly instructions. Depending on the algorithm, implementation details and compilation flags the c++ compiler tries to compile ideal assembly instruction sets. SIMD instructions can only be used when there is a contiguous memory access, thus in some cases a poor memory layout choice may lead to a much lower flops.

For most modern CPU chip architectures AVX is the fastest SIMD instruction set available. The common AVX2 enables the processing of 8 floating point operations per instruction with a CPI (cycles per instructions) of 0.5. The CPI can also vary depending on chip architecture, but to our knowledge all relevant CPU's from figure 1 indeed support a CPI of 0.5 along with AVX2. A lower CPI results in a higher efficiency, as several instructions can be completed in a single cycle.

We define in equation 5.3.1, a function to estimate the number of gigaflops for a given hardware. We define GHZ as the gigahertz which can be reached by the CPU, meanwhile NF is the number of floats which can be processed simultaneously using AVX. CPI is as mentioned the cycles per instructions for the AVX instruction set. Finally we have np which is the number of processors, where we assume a perfect parallelization, meaning 100% of the code can be parallelized.

$$gflops = GHZ * NF * CPI^{-1} * np \tag{7}$$

Since we have peak flops benchmarks available for the NVIDIA P100 and V100s we do not need to make any estimations on the GPU side.

5.3.2 Estimating Arithmetic Intensity

Arithmetic intensity is measured in FLOPS per byte or the number of floating point operations which are computed per byte loaded from memory. The count left part from the cut algorithm (line 6) in figure 2 can be translated to the following isolated c++ code:

Listing 1: Counting the particles left of a cut plane

```
for(auto p= startPtr; p<endPtr; ++p) nLeft += *p < cut;
```

Where p is a C-style array which stores the particles position and $nLeft$ stores the number of particles which are smaller than cut. We can list the operations per loaded float:

1. Compare particle to cut
2. Add result to $nLeft$
3. Increment pointer p
4. Compare pointer p with *endPtr*

Which results in a total of 4 operations. Since a single float is stored using 4 bytes, this computes to 1.0 operations per byte or an arithmetic intensity of four.

Estimating the arithmetic intensity for the GPU is a lot more complicated as it can vary a lot depending on the specific implementation details. But for now we will just assume the same arithmetic intensity as we have had for the CPU.

Note that SIMD instructions do not influence the arithmetic intensity, as the number of floating point operations per byte remains the same. We simply perform a set of operations concurrently. What is however influenced by AVX, is the maximum number of GFLOPS which can be processed by the hardware. If we were to ignore AVX, the algorithm would clearly not be bound by memory, as its performance would lack behind the memory bandwidth. For this reason it is important to consider SIMD instructions.

5.3.3 The Plots

Lets us compute the maximally achievable gflops for Piz Daint. To do so, we plug in the corresponding datapoints from figure 1 into the formula 5.3.1 for $np = 1, 2$ and 3.

$$3.8 * 8 * \frac{1}{0.5} * 2 = 121.6gflops \tag{8}$$

$$3.8 * 8 * \frac{1}{0.5} * 4 = 243.2gflops \tag{9}$$

$$3.8 * 8 * \frac{1}{0.5} * 8 = 486.4gflops \tag{10}$$

The results are depicted as horizontal lines in 14. The memory bandwidth is plotted as a line with an equivalent slope. Finally we add a dotted line representing the arithmetic intensity of the Count Left procedure. To interpret the roofline mode, one has to follow the dotted line starting from the bottom. Weather it intersect with line representing the maximal performance or the memory bandwidth first, gives an indication weather the program is performance or memory bound.

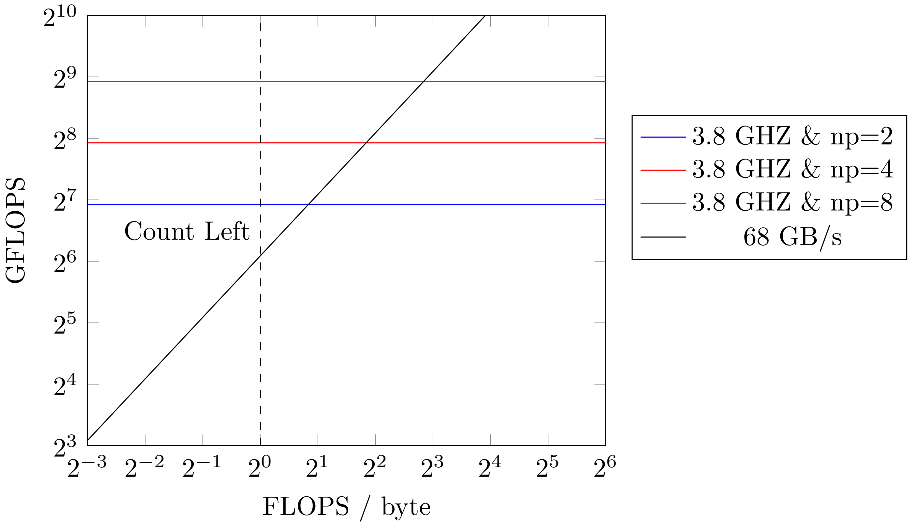


Figure 14: Roofline Model for Piz Daint CPU

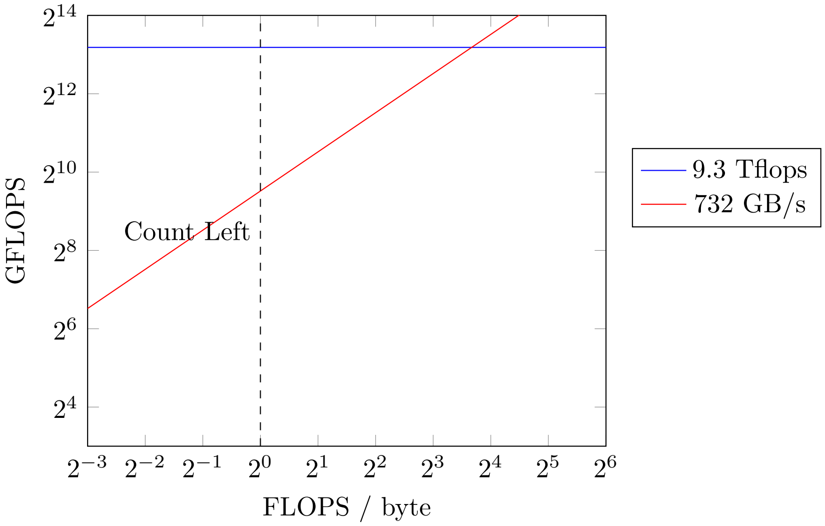


Figure 15: Roofline Model for Piz Daint GPU

As it can be seen in figure 14 and 15 both the GPU and CPU version running on Piz Daint are memory bound, but due to its much higher memory bandwidth limit, a GPU version should be favored.

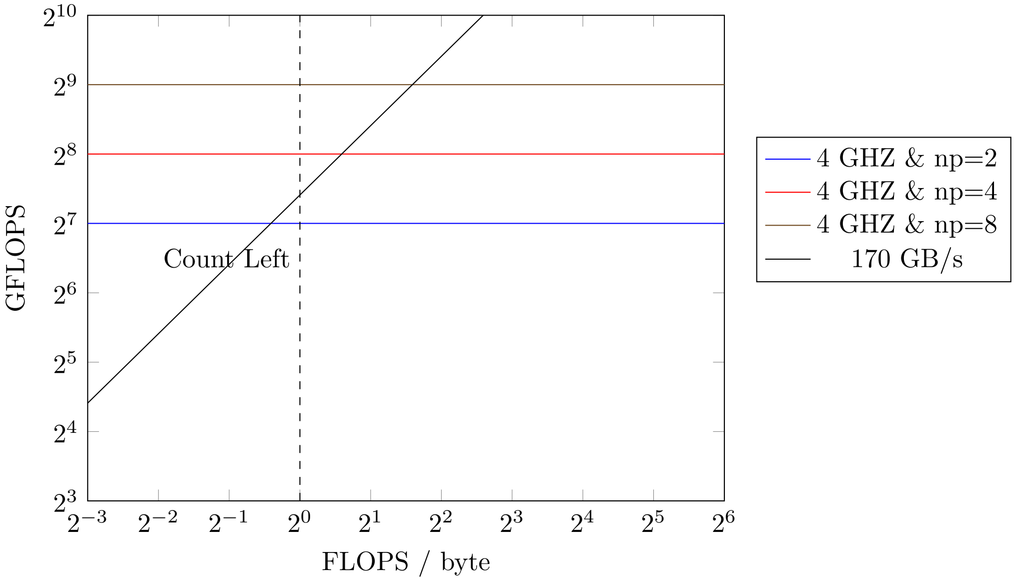


Figure 16: Roofline Model for Summit

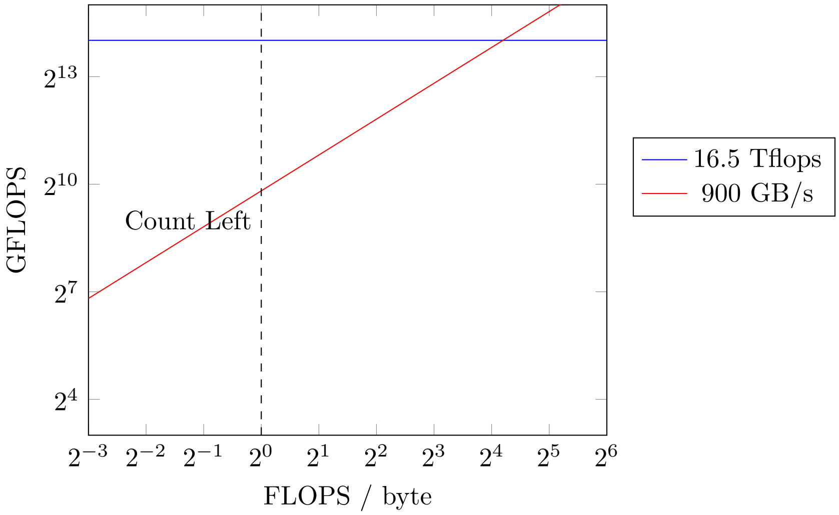


Figure 17: Roofline Model for Summit GPU

As visible in figure 16 and 17 the same applies to Summit. Its notable how the memory bandwidth only becomes the limiting factor when assuming a parallelization with 4 processors.

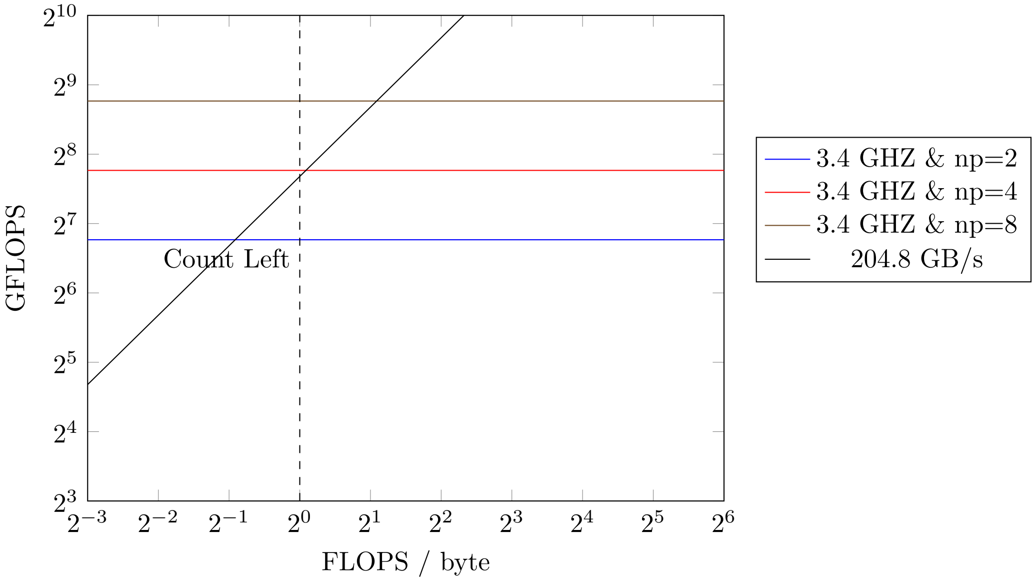


Figure 18: Roofline Model for Alps

Finally alps, which has the highest memory bandwidth compared to its flops, also becomes limited by the bandwidth after using more than four processors.

5.3.4 Empirical Verification

We consider a minimal C++ code to verify the theoretical model. We use the -S flag along with the g++ compiler to generate assembly code from the c++ source code and make sure that AVX commands are enabled. For testing purposes all entries in the array are set to random values between 0 and 1, and cut is set to 0.5.

In a first test, we do not enable AVX, but turn on $O3$. The generated assembly code is depicted in listing 2.

Listing 2: Reduction Assembler Code without AVX

```
.L18:
   movups    (%rax), %xmm0
   addq   $16, %rax
   cmpltps   %xmm2, %xmm0
   psubd %xmm0, %xmm1
   cmpq   %rdx, %rax
   jne    .L18
```

psubd is a packaged instruction, meaning it already uses some form of SIMD instructions. The command is used in the MME and later the SSE2 instruction set. Since we can observe that the *xmm* registers are used, we know its a SSE2 instruction.

If we additionally set the compile flag -march=native we end up with the instructions as depicted in listing 3.

Listing 3: Reduction Assembler Code with AVX2

```
.L19:
   vmovups   (%rax), %ymm3
   addq   $32, %rax
   vcmpltps %ymm2, %ymm3, %ymm0
   vpsubd    %ymm0, %ymm1, %ymm1
   cmpq   %rdx, %rax
   jne    .L19
```

We can identify *vmovups*, *vcomplpts* and *vpsubd* which are AVX commands. Since they are using the ymm instead of the xmm registers, we know these are AVX2 and not AVX commands.

TODO: Fix, Redo this part on daint The following test results were achieved from a Intel(R) Core(TM) i9-10885H CPU @ 2.40GHz processor. We perform the conditional reduction on 2^{27} particles. The reduction takes 244420 microseconds. This means we have a throughput of $2^{27}/10^{12} * 10^6/24420 = 0.00549 tflops$. This of course does not lie anywhere near the theoretical maximum, which even for a single processor ($np = 1$) is $2.4 * 8 * 2 * 1/1000 = 0.0384 tflops.$. This is a strong indication, that already with a single processor solutions, we are in the realm of bandwidth limited algorithms. This effect of course, would become even stronger when including parallelism.

5.4 Runtime Estimates

We construct runtime estimation for the CPU version and two different GPU implementations. In the first GPU implementation *GPU Counting*, we simply send an array of particles to the GPU, where counting is then performed. In a more advanced variant *GPU Counting and Partitioning* we send the particles from the CPU to GPU only once and perform the partitioning on the GPU as well.

We assume a precision p of 32, which is general standard for both integers and floats and its a sensible assumption for astrophysical simulations. The total storage required for all particle positions is $32 \times 3 \times Nbits = 4 \times 3 \times NBytes = 12 \times NBytes$. Furthermore we assume $d = 1024$ and $N = 10^9$.

5.4.1 CPU Version

Whilst the number of cells increases each time a level is added, simultaneously the number of particles to be iterated over . Thus the amount of cells which are to be iterated cancels out and we end up with the size of all particles s divide by the memory bandwidth B_C.

Each time the leaf cells of the SPTDS are split into two child cells and appended to the tree, we end up with twice the number of leaf cells. Considering we want to end up with d leaf cells in the end, we need to perform the cut algorithm $\lceil log_2(d) \rceil$ times for all leaf cells.

Consequently we sum over all $\lceil log_2(d) \rceil$ iterations, where in each iteration a cut has to be found for i cells. The cut itself is found in p iterations, where the actual array of particles is each time we grow the SPTDS by a level. As only the memory bandwidth is considered, we divide the size of the particles to be iterated over.

$$\sum_{i=1}^{\lceil \log_2 d \rceil} i \times p \times \frac{\frac{s}{i}}{B_C} \tag{11}$$

i cancels out from the equation and we end up with:

$$\sum_{i=1}^{\lceil \log_2 d \rceil} p \times \frac{s}{B_C} \tag{12}$$

Finally we can simplify to:

$$\lceil \log_2(d) \rceil \times \left(p \times \frac{s}{B_C} \right) \tag{13}$$

5.4.2 GPU Counting

The Equation 14 for the GPU is similar, the only difference being that we use the GPU memory bandwidth B_{GPU} instead of the CPU bandwidth. Furthermore we have to consider the time it takes to send the data from the CPU to the GPU. This adds the terms size divided by CPU to GPU memory bandwidth denoted as I_{GC}. Finally we also need to load the data from the CPU memory to the CPU before we are able to send it.

$$\lceil log(d) \rceil \times \left(p \times \frac{s}{B_G} + \frac{s}{I_{GC}} + \frac{s}{B_C} \right) \tag{14}$$

5.4.3 GPU Counting and Partitioning

We can also consider performing the partitioning on the GPU, this means that there is no need to send data from the CPU to GPU each time we want to find a cut, allowing us to reduce the costly overheads. This means that we are able to move the corresponding terms out of the brackets.

$$\lceil log(d) \rceil \times \left(p \times \frac{s}{B_G} \right) + \frac{s}{I_{GC}} + \frac{s}{B_C} = t \tag{15}$$

5.4.4 Plugin Values

Let us plugin the values from figure 1 into the corresponding formulas 13, 14 and 15 for Piz Daint. The naive implementation yields the following speeds for the naive CPU implementation:

$$\lceil log(1024) \rceil \times \left(32 \times \frac{12GB}{68GB/s} \right) = 56.47841s \tag{16}$$

And the corresponding GPU implementation:

$$\lceil log(1024) \rceil \times \left(32 \times \frac{12GB}{732GB/s} + \frac{12GB}{32GB/s} + \frac{12GB}{68GB/s} \right) = 10.762s \tag{17}$$

This yields in a speed-up of:

$$\frac{56.47841}{10.762} = 5.24794\times \tag{18}$$

When considering the GPU Counting and Partitioning we end up with: And the corresponding GPU implementation:

$$\lceil log(1024) \rceil \times \left(32 \times \frac{12GB}{732GB/s} \right) + \times \frac{12GB}{32GB/s} + \frac{12GB}{68GB/s} = 5.79802s \tag{19}$$

This yields in a speed-up of:

$$\frac{56.47841}{5.79802} = 9.74097\times \tag{20}$$

5.5 Conclusion

The computations for Eiger and Summit can analogously to the ones from Piz Daint. For Eiger we cannot make any assumption for a GPU version as its CPU only system. All the results are plotted in 19. As expected the GPU Counting outperforms the CPU version on both hybrid (CPU and GPU available)super computers. Furthermore GPU Counting and Partitioning yields more performance improvements, but the speed up is less drastic. As the model may differ from reality depending in implementation and compilation details, these results cannot be taken for granted and we will compare the results with actual empirical observations in section 8.

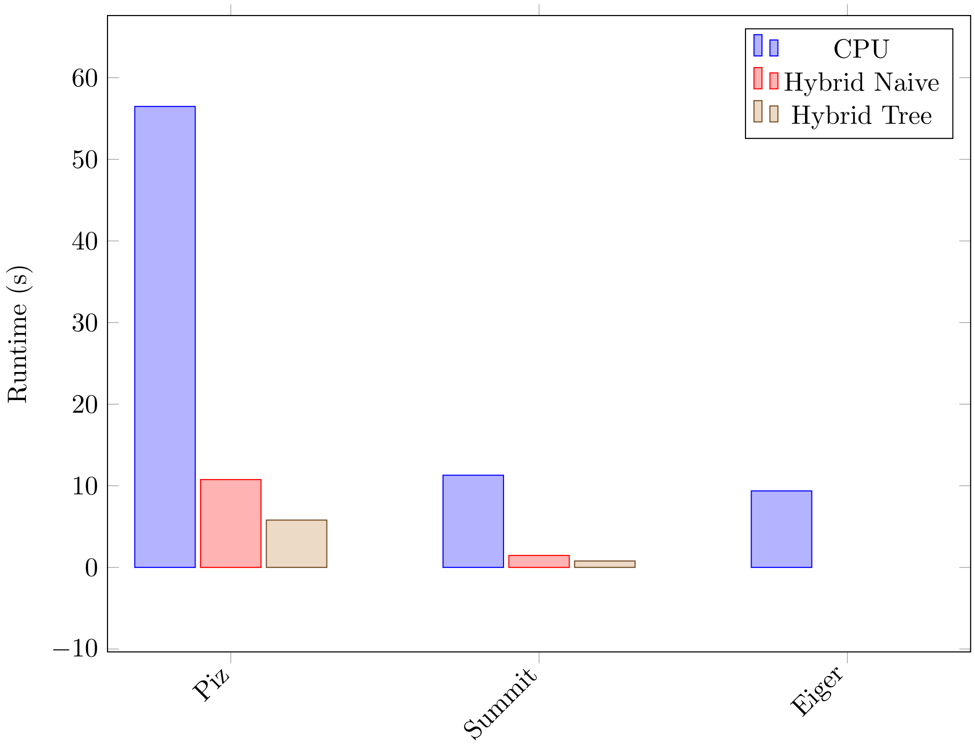


Figure 19: Execution times of different strategies

6 CPU Implementation

In this section we describe the stand-alone implementation of the ORB algorithm using CPU acceleration only. Building on the CPU version, we later introduce GPU accelerated alternatives to selected parts of the code.

We develop somewhat separately from the PKDGrav codebase in order to keep the project smaller and more easy to understand. However we make use of the machine dependent layer (MDL) from PKDGrav, which is used to distribute the workload among processors. This reduces the complexity of the implementation and we do not have to worry about communication methods and CPU parallelism too much. Furthermore it makes integration into PKDGrav simple. Note that processors might be located in different nodes, but MDL is general enough to abstract node and processor level communication and parallelization away.

We refrain from describing each part of the code in detail, however any details crucial to performance are described. Furthermore, we will pay special attention to the CUDA kernels and also memory management strategies for several reasons: minor implementation details can have a big impact upon the performance and increased replicability.

For this project we will use C++ 17 along with CUDA 11.3, OpenMPI 4.1.0 and Blitz++ 1.0.2 [12].

6.1 MDL and PKDGrav

MDL, the machine dependent layer provides an abstraction layer around the parallel primitives introduced by PKDGrav. The Master layer is responsible to coordinate the flow of the program. It does so by calling the PST (Processor Set Tree), which distributes the tasks and gets processors to work on them. As the name suggests, the PST is organized as a binary tree where intermediate nodes contains a pointer to a consecutive processor as well as a next lower node. Each leaf cell of the graph then correspond to an individual processor.

Parallel processes are dispatched by descending the PST until they reach all processors. There computations are performed and the results are combined by passing them back up the PST until they reach the master. A parallel process can be programmed by implementing the service interface as explained in section 6.5.

Each processor has its own local data, which can be accessed by calling $pst- > lcl$.

6.2 Particles

The particles array has ownership over all particle objects. It has a (N,3) shape where each row represents a single particle. To enable coalesced memory access when iterating over a single axis i.e. $x_{:,axis}$, we choose a row major storage order, over a column major storage order. Even though column accesses patterns i.e. $x_{i,:}$ are used as well, they happen less frequently. Most importantly there are cases where entire section of a row have to copied, which is by magnitude faster with row major storage.

We use C-style arrays interchangeably with Blitz++ arrays to store all particles. Blitz++ is an open source wrapper class around C-style arrays, which helps with pointer management and can speed up the debugging process by keeping track of the array boundaries. Furthermore it features array slicing and does its own memory management. In certain cases we need to access the actual C-style array, which can be accessed with $array.data()$. For example when iterating over the Blitz++ array, we have observed compiled assembly code which does not reflect the latest SIMD instructions.

The particle data is loaded into the local data storage of each processor, meaning each processor owns a different unique set of particles.

6.3 Cell

The cell class is a structure keeping track of the fundamental cell information. In essence it is the analogue to the concept of the *cell* which we have already introduced in section 4.

Quickly building and accessing elements of the SPTDS generated by ORB requires a suitable data structure. Instead of a tree with pointers, we can use a heap since the necessary conditions for a nearly complete binary tree are met as seen in section **??**. Elements within a heap can be accessed and added in constant runtime. Depicted in figure 20 is the finished SPTDS of the sample dataset stored as a heap.

The SPTDS and thus the cells are constructed on the master alone, subsets cells are only distributed along the SPTDS when launching certain services.

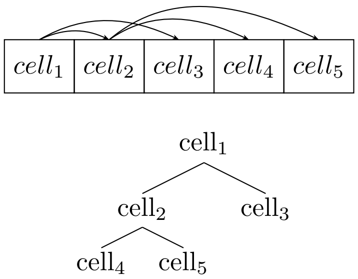


Figure 20: Tree as heap

An array can be used as an efficient storage medium for the heap, since in our case d can be set at compile time and the maximum number of cells in the SPTDS can be derived from that. Meaning the array can be allocated statically. Because MDL communicates data as arrays between the threads, another advantage of the array storage is the absence of any costly data structure conversion.

Since the STPDS is only a nearly complete binary tree and not a complete binary tree, we have to be careful when iterating over the levels of three, as the very last level is not filled entirely with cells. But because d is known, many attributes of the SPTDS can be determined deterministically.

6.3.1 Heap Conditions

As with all heap data-structures we have the following conditions:

- The root cell has index 1
- The left child of cell at index i can be accessed at the index $i \times 2$.
- The left child of cell at index i can be accessed at the index $i \times 2 + 1$.

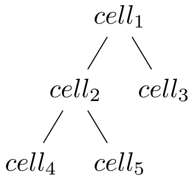


Figure 21: Tree with $d = 3$

We can derive the following constraints given the number of leaf cells d.

- We can compute the depth of the tree with: $\lceil log_2(d) \rceil$
- The number of leaf cells on the second last level is given by $2^{depth} - d$
- There are exactly $2 \times d - 2^{depth}$ items (which must all be leaf cells) on the last level.
- The total number of cells are $2 \times d - 1$.

In the example tree depicted in figure 21 we indeed observe that the depth corresponds to: $\lceil log_2(d) \rceil = 2$, the number of leaves on the second last level is $2^2 - 3 = 1$ and the number of leaves on the last level are $2 \times 3 - 2^2 = 2$. Finally the total number of cells is $2 \times 3 - 1 = 5$.

6.3.2 Class

The cell class consists of the following variables:

- *id*: Unique identification of the cell instance. Corresponds to its index in the heap array plus one. The plus one comes from the different indexing for heap constraints, where we start with 1, and the classic array indexing starting with 0. It can be used to compute indices of both child cells and parent cells using the above shown formulas.
- $nLeafCells$: Equivalent to d_{cell} and depicts the number of leaf cells to be found in all of its successors. Its used when building the tree, to track weather a cell needs to be split or not.
- *lower*: 3D point coordinate representing the lower boundary corner of the 3D volume V_{cell} which is encompassed by this cells.
- *upper*: Represents the upper boundary corner of the volume.
- $cutAxis$: Stores the axis where the cut plane is to be searched for. **TODO:** LOD??

6.4 Mapping Cells to Particles

As SPTDS heap is stored on the master but the particles are distributed among the local storages of the processors, its is crucial to know which particles are encompassed in the volume of which cell. Thus we introduce a data structure to keep track of the relation between cells and its particles. Thanks to the partitioning algorithm all particles belonging to a single cell can be found in a consecutive section of the particles array. Thus we can use a 2D array, which again is stored in the local data of the processor, where for each cell the corresponding range is stored as a tuple. Making the size of the array equivalent to the total number of cells $2 \times d - 1$ multiplied by two. Naturally the data structure needs to be updated across all processors whenever we a partition is performed.

6.5 Services

A service implementation consists of a $Service()$ function as well as a $Combine$ function. The service is handed two a PST class which stores information about the process location as well as the local data. Furthermore two void pointers storing providing storage for the input and output data called in and out along with their respective sizes. The reason void pointers are used, is the actual data structure of the input is variable for each service and this solution provides enough flexibility. Data stored inside the void pointers can be converted into their respective classes using casting. The finished results of the computations are then stored inside the output pointers.

In the $Combine()$ function two output void pointers $vout$ and $vout2$ and the sizes of the underlying arrays are given as parameters. A combination strategy of the elements can be chosen and implemented, where the results is to be stored in $vout$. For example if we were to implement a service which returns the sum of all particles, the combine function would simply store the sum of $vout$ and $vout2$ in $vout$.

6.5.1 Init and Finalize Service

All data stored as local data, which is not managed by Blitz++, must be allocated when initializing and after the program has finished, the memory is freed to avoid any leaks. Allocation and freeing is not free of cost, thus it makes sense to reuse memory whenever possible.

The initialize service is also responsible to load the particle data into the local memory of each PST.

6.5.2 Count Left Service

As mentioned counting the elements smaller than the cut position, or as we call it the count left procedure is computationally intensive. For this reason we introduce a service which along with PST distributes the task among the processors. For the $Service()$ function which performs the actual computations, in points to an array of cells and as we know the length of the array, we can simply iterate over all of its elements and cast them to the correct cell data structure as shown in listing 4 on line 1-2. In case the $foundCut$ flag of cell was set to true, we know that the ideal cut was already found, thus we can continue with the next cell (line 4-6). We then read the begin and end indices of the particles array corresponding to the range of objects contained within the cells volume and respectively make a slice of the local particles array (lines 7-11). We then start and end pointers and obtain the cut position to be tested (lines 13 -17). We then count the number of particles smaller than the cut and write the result in the proper index of the output arrays (lines 18-23).

Listing 4: Part of the Count Left Service() method

```
1 for (int cellPtrOffset=0; cellPtrOffset<nCells; ++cellPtrOffset){
2    auto cell = static_cast<Cell>(*(in + cellPtrOffset));
```

```

    if (cell.foundCut) {
        continue;
    }
    int beginInd = pst->lcl->cellToRangeMap(cell.id, 0);
    int endInd =  pst->lcl->cellToRangeMap(cell.id, 1);

    blitz::Array<float,1> particles =
    pst->lcl->particlesAxis(blitz::Range(beginInd, endInd));

    float * startPtr = particles.data();
    float * endPtr = startPtr + (endInd - beginInd);

    int nLeft = 0;
    float cut = cell.getCut();
    for(auto p= startPtr; p<endPtr; ++p)
    {
        nLeft += *p < cut;
    }

    out[cellPtrOffset] = nLeft;
}
```

Combining the data is straight forward, as in *Combine*() *vout* and *vout*2 contain the counts for a given number of particles. As the master is interested in a global count across all the particles contained within all local storages, we simply sum each element of *vout* together with *vout*2.

Listing 5: Part of the Count Left Combine() method

```
    for(auto i=0; i<nCounts; ++i)
        out[i] += out2[i];
```

6.5.3 Count

Counting the total number of particles for each cell is trivial with the *cellToRange* map, as we can directly know the result when subtracting the begin index from the end index of each cell. The *Combine*() is equal to the one shown for the Count Left Service.

Listing 6: Part of Count Service() method

```
    for (int cellPtrOffset=0; cellPtrOffset<nCells; ++cellPtrOffset) {
        auto cell = static_cast<Cell>(*(in + cellPtrOffset));
        out[cellPtrOffset] =
        lcl->cellToRangeMap(cell.id,1) -
        lcl->cellToRangeMap(cell.id,0);
    }
```

6.5.4 Partition

The Partitioning Service is a direct implementation of the hoare partition and can be seen in 7 on lines 1-29. Interesting about the code bit are lines 31 to 37 where the CellToRange map is updated for the pair of children of the divided cell **TODO:** extend

Listing 7: Partition Service

```cpp
for (int cellPtrOffset=0; cellPtrOffset<nCells; ++cellPtrOffset){
    auto cell = static_cast<Cell>(*(in + cellPtrOffset));

    int beginInd = pst->lcl->cellToRangeMap(cell.id, 0);
    int endInd = pst->lcl->cellToRangeMap(cell.id, 1);

    int i = beginInd-1, j = endInd;
    float cut = cell.getCut();

    while(true)
    {
        do
        {
            i++;
        } while(lcl->particles(i,cell.cutAxis)<cut and i<=endInd);

        do
        {
            j--;
        } while(lcl->particles(j,cell.cutAxis)>cut and j>=beginInd);

        if(i >= j) {
            break;
        }

        swap(lcl->particles, i, j);
    }

    swap(lcl->particles, i, endInd -1);

    lcl->cellToRangeMap(cell.getLeftChildId(), 0) =
    lcl->cellToRangeMap(cell.id, 0);
    lcl->cellToRangeMap(cell.getLeftChildId(), 1) = i;

    lcl->cellToRangeMap(cell.getRightChildId(), 0) = i;
    lcl->cellToRangeMap(cell.getRightChildId(), 1) =
    lcl->cellToRangeMap(cell.id, 1);

}
```

6.5.5 Make Axis

As mentioned in section 2.1 the relevant axis to search for the cut position differs for each cell. Therefore we cannot simply iterate over a single array, we need to iterate over the relevant axis instead. In order to simplify the process we introduce a service which copies iterates over all cells and copies the slice of coordinates which are encompassed in its volume and lie on its cut axis to a temporary array.

6.6 Parallel Schedule

In the context of parallelization, we define the number of processors as *np*. Initially we assume that each processor has a random unique subset of all the particles stored in its local memory, this has two reasons. For one we run into memory limitations quickly when trying to load all the particles onto a single node, furthermore memory bandwidths limitations can be multiplied by the number of processors and are thus a lot higher.

In the running example this might look as follows:

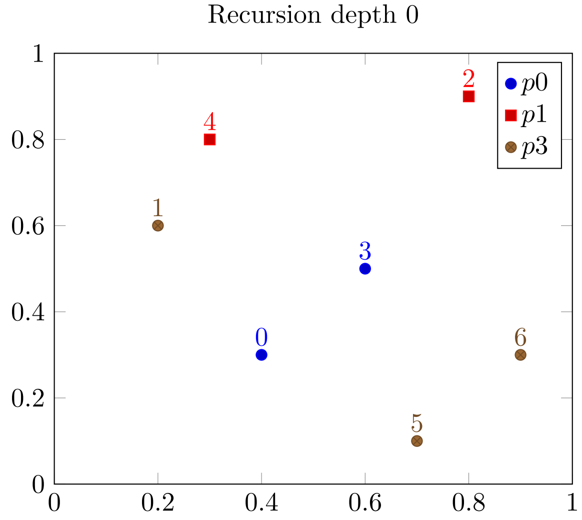


Figure 22: Example particles distributed randomly across 3 nodes

The Master schedules all other processors but also performs tasks by itself. The exact schedule is illustrated in figure 23 where a service which is dispatched from the master and executed on all processors is depicted as a horizontal rectangle. Computations which are only performed by a single processor are represented by a vertical rectangle.

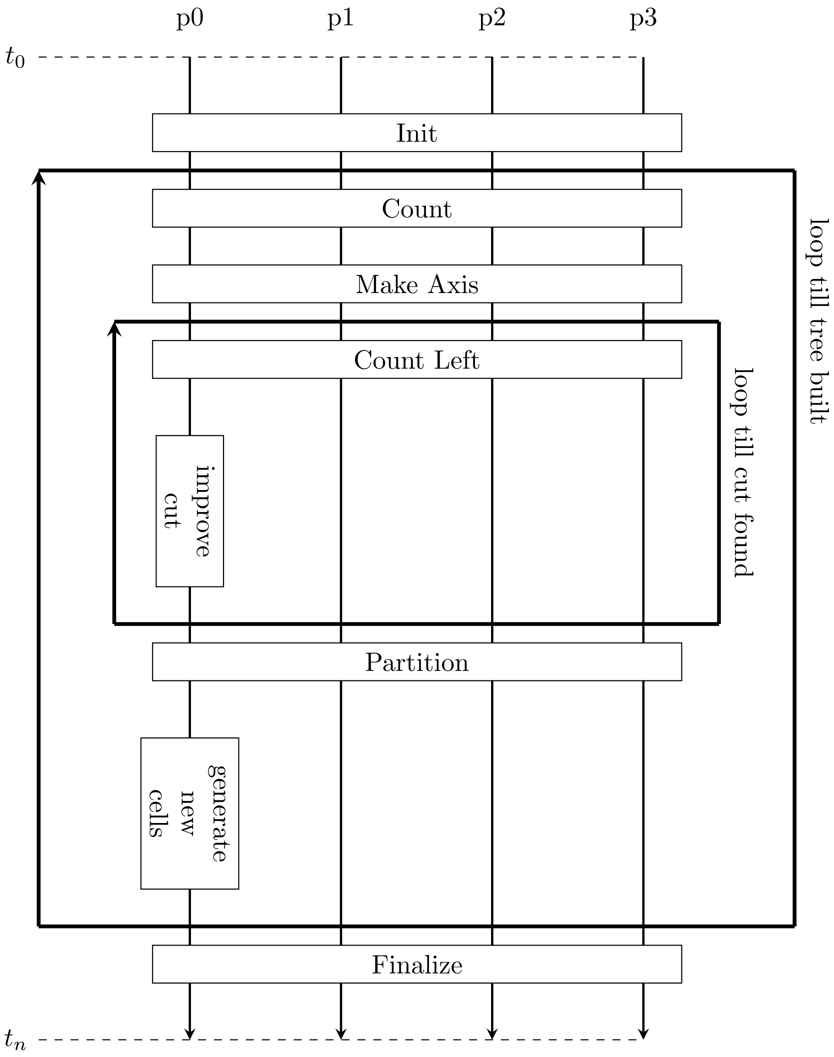


Figure 23: Parallelized ORB CPU version

7 GPU Implementation

CUDA is a parallel computing platform and programming model invented by Nvidia. It allows software developers and programmers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. The CUDA platform is designed to work with programming languages such as C, C++, and Fortran and gives direct access to the GPU's virtual instruction set and parallel computational elements.

7.1 Relevant CUDA Concepts

CUDA gives us the ability to launch kernels, which are written in C with some additional CUDA specific syntax. These kernel can be run from the CPU, commonly refereed to as the host, and are then executed on the GPU, also known as the device. The device and the host have a separate memory and as seen in section 5.1 the transfer I_{CG} rates between the GPU and CPU are generally slower, under performing I_C and I_G speeds. Therefore one of key the challenges when rewriting CPU code to GPU code is to limit data transfers between the GPU and CPU as much as possible. Furthermore we have divide our problems into subproblems, where each subproblem is then executed on a block. CUDA cannot give any guarantees considering the order of execution of these blocks, therefore we have to fundamentally rethink algorithms when porting them from CPU to GPU code, where some algorithms are more or less fitted. Generally we only think about problems which are applied on large data arrays, the less connection between the individual results exists, the easier it is to implement an GPU version of the computation. Furthermore we can assume that problems with less branching work better on the GPU due to various reasons, which we will explain in detail later.

Building a tree is therefore a rather challenging problem as there are many computations which influence another and building a tree involves a lot of branching, at least when it is done in a conventional manner.

Listing 8: Calling a CUDA Kernel

```
1 add<<<
2     nBlocks,
3     nThreads,
4     nThreads * bytesPerThread,
5     stream
6 >>>(
7     param1,
8     param2
9 );
```

A kernel is executed exactly once for every kernel, where a block consists of many kernels. The number of kernels per block and the total number of blocks can be defined by the user as seen in the kernel call syntax in figure 8 on line 2 and 3.

7.1.1 Warps

Each consecutive grouping of 32 threads form a warp. All threads within a warp are executed in parallel, given that there is no warp divergence. Because warps are executed in parallel, there is no need for synchronization between the threads of the same warp. A warp divergence can occur, if there is a control statements, where two or more threads from the same warp execute a different code. This leads to a decrease in performance and should be avoided whenever possible.

7.1.2 Memory

Each block has access to shared memory register, where the capacity of this register can be defined at runtime by the host as seen in figure 8 on line 4. The maximum shared memory size depends on the hardware but usually there are 48kB per block available. There are also upper limits for the number of threads per block, usually **TODO:** B or b?? around 512 to 1024. With a shared memory size of 48kb and 1024 threads we get $\frac{48000b}{1024} = 46b$ per thread. This is equivalent to a little bit more than one float per thread. However usually we do not max out the thread counts per block, leaving us with more shared memory space per thread.

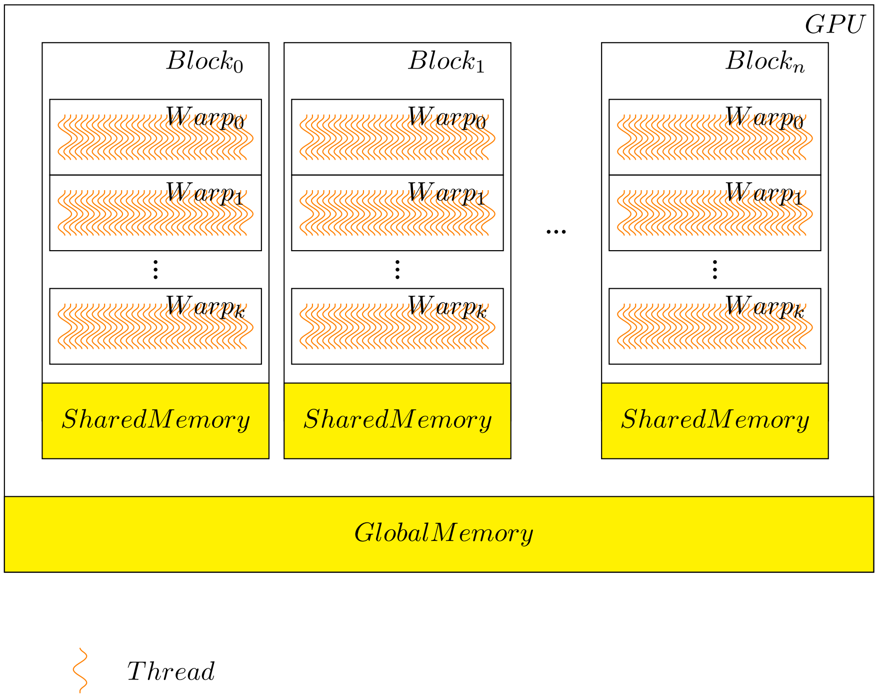
Data in the shared memory exists only as long as the kernel exists and it cannot be accessed across different blocks. Race conditions do apply to shared memory, but threads in a block can be synchronized, therefore it is not safe to have writes from multiple threads on the same shared memory address. There exists a CUDA command $\_\_synchtreads()$ which allows us to synchronize all threads within a block, enabling us to control the execution order of individual statements.

Additionally to the shared memory, there exists global memory. Global memory is persistent even after a block or a kernel has finished and is the only way to send data between the device and the host. Since the data is not deleted after a kernel has finished, we can and should reuse the data whenever possible for other kernels, reducing the memory transfer overhead. Global memory is significantly slower than shared memory and it is best practice to copy data from global memory to shared memory before performing actual computations on it, thus reducing the total number of memory accesses on global memory. After the computations have finished, the results can be written back from shared to global memory.

7.1.3 Memory Access Patterns

Global memory is fetched from memory in 32 Byte packages, which translates to $\frac{32}{4} = 8$ floats. A simple aligned sequential memory access pattern, where each tread reads a single float from global memory, results in 4 transaction per warp (32 threads). Unaligned memory access can have minor performance penalties because more memory banks have to be loaded per warp.

For shared memory each bank consists of 32 bits, which translates to a 32 bit precision number. If 32 unique threads access 32 different banks, then its considered an ideal access pattern, because the banks can be loaded in parallel. If however multiple threads access the same memory, a bank conflict is introduced and the access speed is greatly reduced.



7.1.4 Asynchronous Operations

There are two types of engines that can be used to execute kernels in CUDA streams: copy engines and kernel engines. Copy engines are used to copy data between host and device memory, and between different types of memory on the device. Kernel engines are used to execute CUDA kernels. The number of individual engines depends on the actual hardware, lower end GPUs usually have a single kernel and a copy engine, more advanced architectures can have more than one copy or kernel engine.

A CUDA stream is a sequence of commands that are executed in order on a CUDA device. Streams can be used to improve the runtime of a CUDA program by overlapping the execution of different kernels. For example, a copy kernel can be executed in one stream while a compute kernel is executed in another stream. This overlap can lead to a significant performance improvement.

7.1.5 Synchronization

Whilst threads within a block can be synchronized, this is not the case for the blocks launched from a kernel. The only real synchronization technique, is to wait for a kernel, meaning all blocks associated with this kernel, to finish and execute a consecutive task using another kernel.

Since the GPU is limited in its computing and also memory capabilities, the number of blocks which are run in parallel are limited, therefore CUDA cannot give us any guarantee, weather a set of blocks are run serially or in parallel. Some of the blocks might be run in parallel, meanwhile others are run serially. This very constraint, forces the programmer to rethink algorithms and think of ways, how individual subproblems can be run somewhat independently of each other.

There exists however a way to communicate between blocks in a safe way. Atomic operations can be performed on global memory without any race conditions.

7.2 Streams

As described in section 7.1.4, we can leverage streams to improve the runtime a CUDA accelerated application. The way we make use of streams is simple, each thread performs all its computations using on a unique stream. Since leveraging streams only improves the runtime in low numbers because the actual copy and kernel engines are not very numerous, this variant is very simple and effective. We have observed that using more streams on the individual threads does not improve the runtime, quiet contrary it can have a negative impact.

7.3 Memory Management

We can use several strategies in order to improve the performance of memory allocation:

1. **Pinned Memory** As the GPU cannot access the default paged memory directly, when copying memory from the host to the device, the memory must first be copied to pinned memory. This means if we use pinned memory directly with $cudaMallocHost()$, data transfers can be around twice as fast. In our implementation we use pinned memory for all data, which need to be either sent from the host to device or vice versa.
2. **Reuse Memory** We can avoid allocation and freeing commands altogether and instead reuse previously allocated memory whenever possible. In our implementation we allocate the memory in the very beginning, whereas we have a fixed number of particles and a fixed upper limit for the number of cells. This is true for both the CPU and GPU memory.

7.4 GPU Accelerated Count Left

We have described a CPU implementation of the countLeftService. In this section we explain how we can port this problem to the GPU. In essence we can use a general reduction as a basis version, and adapt it to sum up up elements which fulfill a certain condition, where the condition is being smaller than a given value.

Optimized reductions in CUDA are well documented and explained by the NVIDIA developer team. We make use of a reduction introduced by **TODO:** Reference and adapt it slightly to fit our needs. We will reiterate over the important optimization aspects and finally point out the specific adaptations we have made.

We split the array of particles which are contained within one cell into smaller slices, where each slice fits one block. Then the subproblems or blocks can be run independently from each other, in other words individual blocks can be executed in parallel or also serially without any conflicts or need for block level synchronization. After running the conditional summation on each slice of the array, we end up with a an array of results. These results can then be summed together on CPU using a simple iterator. Alternatively we could invoke another sum reduction kernel for this task.

7.4.1 Schedule

The entire schedule of the ORB is depicted in figure 24. In a first step we call the initialize service, where the necessary data is allocated on all devices and the particles are loaded. Furthermore the initial SPTDS is constructed, which essentially only consists of the root node, encompassing the entire domain and all the particles. Next we enter the main loop of ORB, it iterates until it has constructed a SPTDS with the desired size. Within the loop the count service is called, computing the number of cells for each cell summed over all the threads in the system. To prepare the data for the GPU transfer we make the temporary array using the make axis service, which is then sent to the GPU. We can now start with the root finding process, where we count the particles left of the initial cut position using the Count Left Service. Depending on the outcome, we then improve the cut position and repeat until a nearly perfect cut is found for all the leaf cells of the current SPTDS. We now generate two new child cells using the computed cut positions for all leaf cells of the current SPTDS and enrich the current SPTDS with the newly generated cells. Finally the particles array is partitioned accordingly. The tree building loop is then repeated or if the desired size of the SPTDS is reached, we exit the loop and call the finalize service to free the allocated memory.

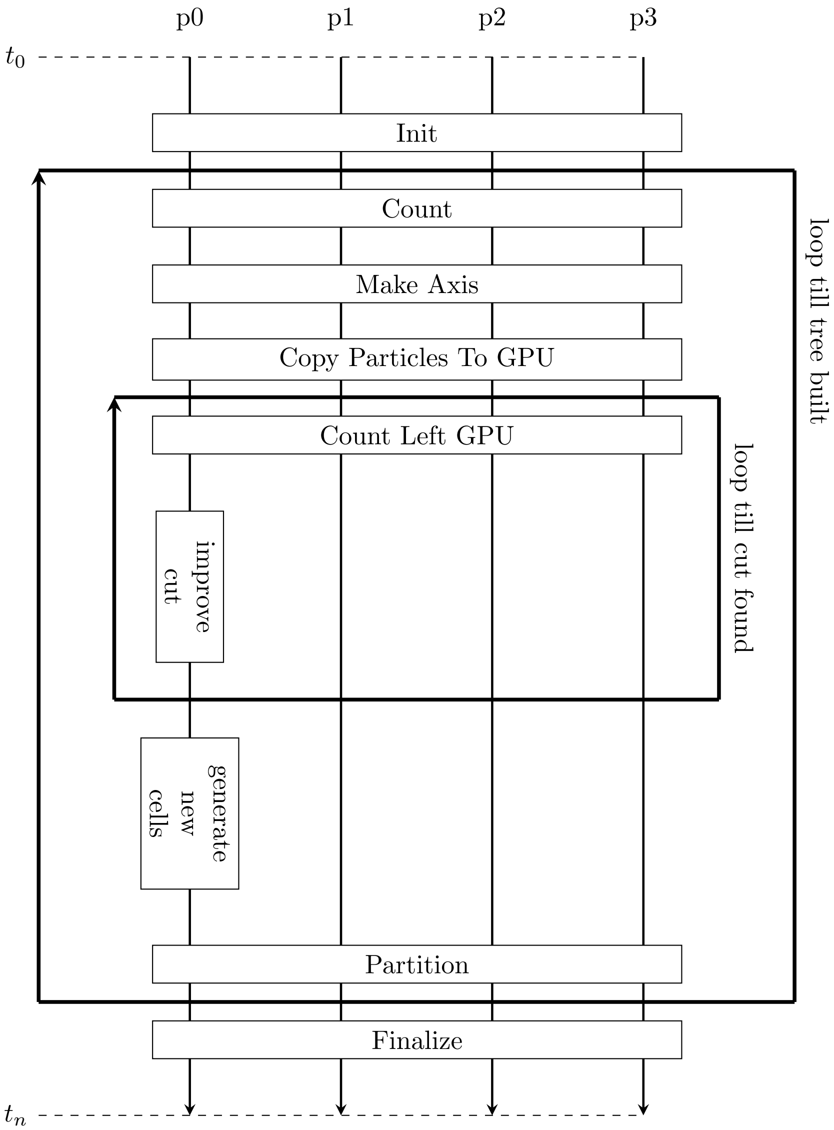


Figure 24: Parallelized ORB GPU version

7.4.2 Service

The GPU version of the Count Left Service invokes the reduction kernel exactly once for every leaf cell of the current SPTDS. Meaning on level 0 of the tree there is only one kernel per thread being executed, when proceeding to further levels, the number of kernels is equivalent to 2^{level}. The number of blocks can be determined by the number of particles in this cell n_{cell} divided by the number of threads per block which is set to 256 and finally we divide this further by the number of elements per thread r.

Since we can pass all the relevant information regarding a cell as input parameters to the cell, there is no need to copy any additional data from the CPU to the GPU. We only need to copy back the results from the GPU to the GPU.

7.4.3 Kernel Code

The actual kernel code is depicted in figure 9. The input parameters of the reduction kernel (line 13-16), the input data, which is in sequential order the particles, the output array, which is the results of the kernel, the cut position and finally the total number of elements contained within the cell. We will

- **Leverage Shared Memory** In the code on line 17 we invocate the shared memory, where is equivalent to the block size, which corresponds to a single float per thread. In a next step (lines 24-27) we apply our condition and copy the results to shared memory. This allows us to work exclusively with shared memory, reducing the amount of costly accesses performed on global memory.
- **Increase Thread Occupancy** There exists an optimum when considering the number of operation performed by a single thread. In order to adapt the number of ops dynamically, we iterate over a certain number of elements in the global memory, as seen in lines 24-27. The input parameter n defines the total number of particles. Therefore if we reduce the total number of blocks by a factor of r, the while loop will iterate over r elements. In our case we set r to 16 as more or less items per thread will result in a lower performance.
- **Meta Programming** The reduction is optimized by using a template, which indicates the total number of thread per block, which is equivalent to the blockSize. Since the template is evaluated at compile time, all the if statements taking blockSize as a parameter in figure do not cause any performance loss. In fact, because we are able to unroll loops, which would be evaluated at runtime, we increase the performance of the code.
- **Avoiding Warp Divergence** Each time we we have an control statement in the code, we introduce a divergence. On the GPU these divergences become especially bad when they are introduced inside a warp. This means that any of the threads of the 32 threads in a warp, have to execute a different code than the rest. For this reason, the *warpReduce* method as on line 2-9 is introduced. We can see on line 32 in figure 9 that this method is only executed when we reach 32 elements in the shared memory, which remain to be added together. Inside the warp reduce method, we again encounter an unrolled loop, but in this case we have no if statements that could generate a divergence. The if statements which are present, are as mentioned, evaluated at compile time only. Inside a warp we are also not dependent on synchronization, thus the thread synchronization call can be omitted as well. Note that this implementation will perform many unnecessary operations, but this does not affect the performance negatively.

Listing 9: Conditional Reduction in CUDA

```
1 template <unsigned int blockSize>
2 extern __device__ void warpReduce(
3    volatile int *sdata, unsigned int tid) {
4    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
5    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
```

```
   if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
   if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
   if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
   if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

template <unsigned int blockSize>
extern __global__ void reduce(
   float *g_idata,
    unsigned int*g_odata,
    float cut,
    int n) {
   __shared__ int sdata[blockSize];

   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*(blockSize) + threadIdx.x;
   unsigned int gridSize = blockSize*gridDim.x;
   sdata[tid] = 0;

   while (i < n) {
      sdata[tid] += (g_idata[i] <= cut);
      i += gridSize;
   }
   __syncthreads();

   if (blockSize >= 512) {
      if (tid < 256) {
         sdata[tid] += sdata[tid + 256];
      }
      __syncthreads();
   }
   if (blockSize >= 256) {
      if (tid < 128) {
         sdata[tid] += sdata[tid + 128];
      } __syncthreads();
   }
   if (blockSize >= 128) {
      if (tid < 64) {
         sdata[tid] += sdata[tid + 64];
      } __syncthreads();
   }
   if (tid < 32) {
      warpReduce<blockSize>(sdata, tid);
   }
   if (tid == 0) {
      g_odata[blockIdx.x] = sdata[0];
   }
}
```

We execute the reduction kernel once for each cell, where we distribute the particles within the cell among a set of blocks. This makes the implementation straight forward but as we increase the number of cells, we make make more calls to the reduction kernel. This is problematic because each initialization of a kernel comes with some overhead, degrading the performance gradually with the tree traversal, as the overhead to computation ratio becomes more and more unfavorable.

7.5 Improved GPU Accelerated Count Left

To solve the degrading performance problem we make some changes to the kernel and the overall schedule. The main idea is to prepare the necessary data for all cells in advance, effectively reducing the overall number of necessary kernel calls to $\lceil \log_2 d \rceil$ instead of $2 \times d$. Each block is provided with information concerning the start and end index of the cells particles, the cut position as well as an index of the cell.

We will explain the improved version using a simple example: Let us consider $cell_2$ with a volume that encompasses particles in the range 0 - 10240. $cell_3$ encompasses particles in the range 10240 - 20480. We have a blocksize of 256 threads and the elements per thread are 16, thus each block processes $256 * 16 = 4096$ elements. In this case we reserve three blocks for each cell, where the first block processes elements 0 - 4096 and the second one 4096 - 10240. By assigning more work to the second block, we make sure that there are no underworked threads, as measurements have shown a slight increase in elements per thread is better than a decrease. Furthermore we can avoid blocks where not even all the threads are occupied, because less than 256 elements are processed on the block. Consequently the third and fourth block respectively process particles 10240 to 14446 and 14446 to 20480.

7.5.1 Schedule

Only a single change has to be made to the schedule: We introduce another service which is called GPU Copy Cell Service, which does as the name says, copying the cell information from the CPU to the GPU. Since only the cut positions change while we are iterating over the inner loop, this service can be called outside of it. The Improved GPU Count Left Service copies the cut data each time it is invoked. Other than that, the schedule remains equal.

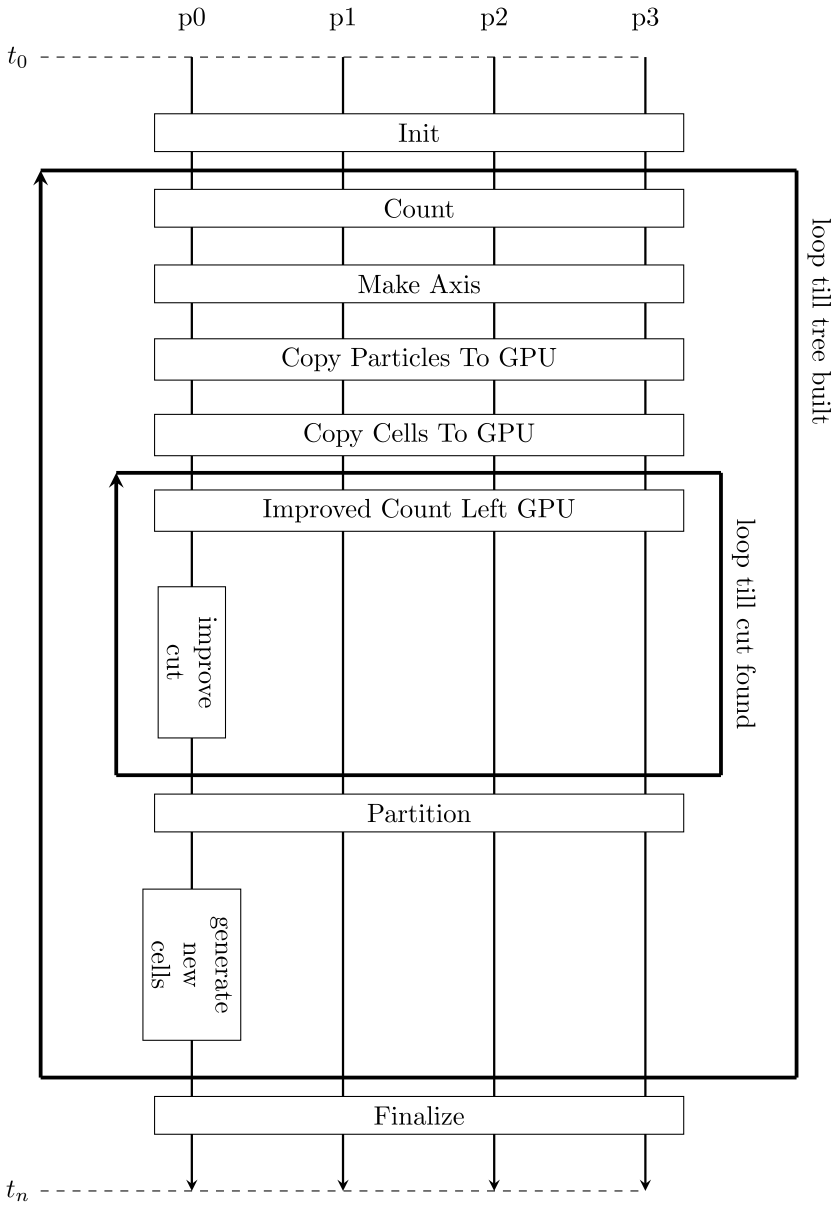


Figure 25: Parallelized ORB GPU version 2

7.5.2 Kernel Code

The kernel code as seen in listing 10 largely remains the same. New data pointers are passed along as a parameter where g_begin and g_end mark the begin and end of the important particle array slice, furthermore g_cuts is an array of cell cuts for each block. As we prepare the data in a way, that each block only needs to operate over the particles contained within a single cell, the data can be read using the blockIdx.x (lines 20 - 22) which is a CUDA variable and provides a sequential indexing of blocks associated with the kernel. Everything else, including the subroutine *warpReduce* remain equal.

Listing 10: Kernel Optimized GPU Count Left

```
template <unsigned int blockSize>
extern __global__ void reduce(
float * g_idata,
unsigned int * g_begins,
unsigned int * g_ends,
float * g_cuts,
unsigned int * g_odata) {
   __shared__ unsigned int s_data[blockSize];

   unsigned int tid = threadIdx.x;
   const unsigned int begin = g_begins[blockIdx.x];
   const unsigned int end = g_ends[blockIdx.x];
   const float cut = g_cuts[blockIdx.x];

   unsigned int i = begin + tid;
   s_data[tid] = 0;

   // unaligned coalesced g memory access
   while (i < end) {
      s_data[tid] += (g_idata[i] <= cut);
      i += blockSize;
   }
   __syncthreads();

   if (blockSize >= 512) {
      if (tid < 256) {
         s_data[tid] += s_data[tid + 256];
      }
      __syncthreads();
   }
   if (blockSize >= 256) {
      if (tid < 128) {
         s_data[tid] += s_data[tid + 128];
      } __syncthreads();
   }
   if (blockSize >= 128) {
      if (tid < 64) {
         s_data[tid] += s_data[tid + 64];
      } __syncthreads();
   }
   if (tid < 32) {
      warpReduce<blockSize>(s_data, tid);
   }
   if (tid == 0) {
      g_odata[blockIdx.x] = s_data[0];
   }
}
```

Figure 26: Reduction in CUDA

7.6 GPU Accelerated Partitioning

The main bottlenecks of the orb algorithm are now the data transfer between the CPU and GPU and the partitioning of the data. Both problems can be resolved by partitioning the array on the GPU directly thus also removing the need to transfer the data back and forth between the CPU and the GPU.

We describe measurements which were done using the reduction in CUDA, we mention how the reduction cannot necessarily be improved a lot and that the data transfer between the GPU and CPU is the main bottleneck now. We mention how building the tree on the does not make a lot of sense. We explain why instead we focus our efforts to implement a GPU reshuffling method, which can improve runtime by reducing the reshuffling costs and also remove CPU GPU transfers between invocations of the binary cut algorithm altogether.

7.6.1 Schedule

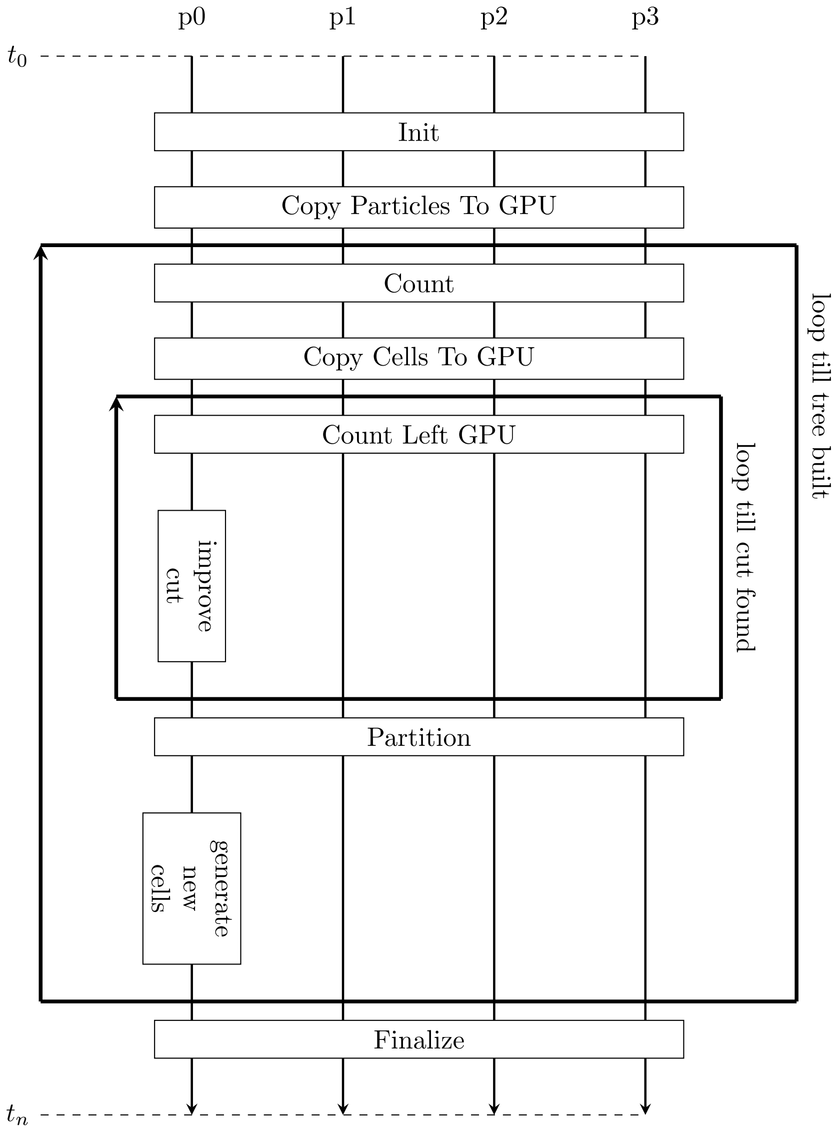


Figure 27: Parallelized ORB with GPU counting and GPU partitioning

We add a timeline which is based on measurements with a CUDA enhanced binary cut algorithm.

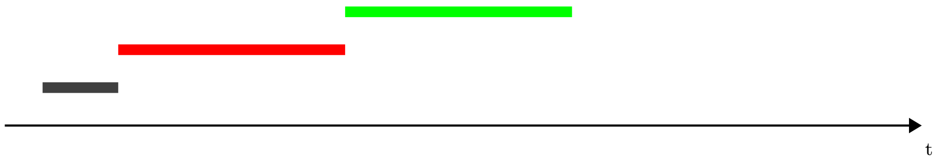


Figure 28: timeline

The partitioning algorithm is a widely used concept and there are various implementations used in different variants of the quick-sort algorithm. The overall design of the implementation is guided by **TODO:** Insert ref and the scan is taken from GPU Gems **TODO:** Insert ref . The main and probably only disadvantage of partitioning the particle array on the GPU, is that it cannot be done in place, thus we need another temporary array allocated on the GPU memory which further restricts the total number of particles. Furthermore we need to store the permutations, because we need to apply them to the other axes as well. We could as well simply perform all permutations from the same kernel at the same time, but this would require 3 temporary arrays, thus reducing the total number of particles by a factor of 2. The solution with the partition array and a single temporary array only results in a reduction of the particles by a factor of $\frac{5}{3}$.

- As we increase the number of cells, we make more calls to the reduction kernel

7.6.2 Code

```
template <unsigned int blockSize>
__global__ void partition(
unsigned int * g_offsetLessEquals,
unsigned int * g_offsetGreater,
float * g_idata,
float * g_odata,
unsigned int * g_permutations,
float pivot,
unsigned int nLeft,
unsigned int n) {
   __shared__ unsigned int s_lessEquals[blockSize * 2];
   __shared__ unsigned int s_greater[blockSize * 2];
   __shared__ unsigned int s_offsetLessEquals;
   __shared__ unsigned int s_offsetGreater;

   unsigned int tid = threadIdx.x;

   unsigned int i = blockIdx.x * blockSize * 2 + 2 * tid;
   unsigned int j = blockIdx.x * blockSize * 2 + 2 * tid + 1;

   bool f1, f2, f3, f4;
   if (i < n) {
      f1 = g_idata[i] <= pivot; f2 = not f1;
      s_lessEquals[2*tid] = f1;
      s_greater[2*tid] = f2;
   } else {
      f1 = false; f2 = false;
      s_lessEquals[2*tid] = 0;
      s_greater[2*tid] = 0;
   }

   if (j < n) {
      f3 = g_idata[j] <= pivot; f4 = not f3;
      s_lessEquals[2*tid+1] = f3;
      s_greater[2*tid+1] = f4;
   } else {
      f3 = false; f4 = false;
      s_lessEquals[2*tid+1] = 0;
      s_greater[2*tid+1] = 0;
   }

   __syncthreads();

   scan(s_lessEquals, tid, blockSize * 2 );
   scan(s_greater, tid, blockSize * 2);
```

Figure 29: Partitioning kernel I

```

    __syncthreads();

    if (tid == blockSize - 1) {
        s_offsetLessEquals =
        atomicAdd(g_offsetLessEquals, s_lessEquals[blockSize*2-1] + f3);
        s_offsetGreater =
        atomicAdd(g_offsetGreater, s_greater[blockSize*2-1] + f4);
    }

    __syncthreads();

    unsigned int indexA = (s_lessEquals[2*tid] + s_offsetLessEquals) * f1 +
    (s_greater[2*tid] + s_offsetGreater + nLeft) * f2;

    unsigned int indexB = (s_lessEquals[2*tid+1] + s_offsetLessEquals) * f3 +
    (s_greater[2*tid+1] + s_offsetGreater + nLeft) * f4;

    if (i < n) {
        g_odata[indexA] = g_idata[i];
        g_permutations[i] = indexA;
    }

    if (j < n) {
        g_odata[indexB] = g_idata[j];
        g_permutations[j] = indexB;
    }
}
```

Figure 30: Partitioning kernel II

```
__device__ void scan(volatile unsigned int * s_idata, unsigned int thid, unsigned int n) {
   unsigned int offset = 1;
   for (unsigned int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
   {
      __syncthreads();
      if (thid < d)
      {
         unsigned int ai = offset*(2*thid+1)-1;
         unsigned int bi = offset*(2*thid+2)-1;
         s_idata[bi] += s_idata[ai];
      }
      offset *= 2;
   }
   if (thid == 0) { s_idata[n - 1] = 0; } // clear the last element
   for (unsigned int d = 1; d < n; d *= 2) // traverse down tree & build scan
   {
      offset >>= 1;
      __syncthreads();
      if (thid < d)
      {
         unsigned int ai = offset*(2*thid+1)-1;
         unsigned int bi = offset*(2*thid+2)-1;
         unsigned int t = s_idata[ai];
         s_idata[ai] = s_idata[bi];
         s_idata[bi] += t;
      }
   }
}
```

Figure 31: Device Scan Kernel

```
template <unsigned int blockSize>
__global__ void permute(
float * g_idata,
float * g_odata,
unsigned int * g_permutations,
int n) {
   unsigned int tid = threadIdx.x;

   unsigned int i = blockIdx.x * blockSize * 2 + 2 * tid;
   unsigned int j = blockIdx.x * blockSize * 2 + 2 * tid + 1;
   //unsigned int gridSize = blockSize*2*gridDim.x;

   if (i < n) {
      g_odata[g_permutations[i]] = g_idata[i];
   }

   if (j < n) {
      g_odata[g_permutations[j]] = g_idata[j];
   }
}
```

Figure 32: Device Permute Kernel

7.7 Integration into PKDGRAV

8 Performance Analysis of ORB

We will use two system for our performance analysis. One is the Summit supercomputer which has been described in section 3.1. The other has the following specs:
AMD EPYC 7702 64-Core Processor, Tesla T4.

8.1 Methodology

8.2 Results

8.3 Comparison to Theoretical Model

8.4 Conclusion

References

[1] Joachim Gerhard Stadel. *Cosmological N-body simulations and their analysis.* PhD thesis, 2001.

[2] G˙Real-time ray tracing on the gpu - ray tracing using cuda and kd-trees. Master's thesis.

[3] Thrust, May 2022. [Online; accessed 28. Jul. 2022].

[4] CUB: Main Page, May 2022. [Online; accessed 28. Jul. 2022].

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2022.

[6] Piz Daint & Piz Dora, April 2022. [Online; accessed 27. Apr. 2022].

[7] ORNL Launches Summit Supercomputer | ORNL, April 2022. [Online; accessed 27. Apr. 2022].

[8] Intel® Xeon® Processor E5-2690 , February 2022. [Online; accessed 27. Apr. 2022].

[9] AMD EPYC™ 7742 | AMD, April 2022. [Online; accessed 27. Apr. 2022].

[10] NVIDIA Tesla P100: The Most Advanced Data Center Accelerator, April 2022. [Online; accessed 27. Apr. 2022].

[11] NVIDIA V100 | NVIDIA, April 2022. [Online; accessed 27. Apr. 2022].

[12] blitzpp. blitz, May 2022. [Online; accessed 11. May 2022].

List of Figures

List of Tables